PATENT ABSTRACTS OF JAPAN

(11)Publication number : 07-271738
(43)Date of publication of application : 20.10.1995

(51)Int.Cl. G06F 15/16
G06F 9/46

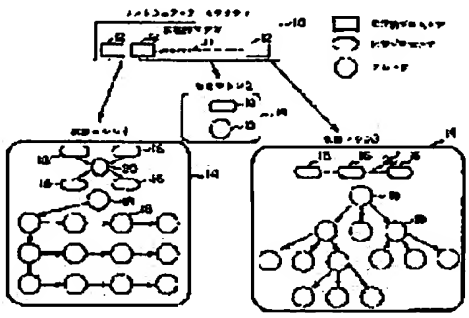
(21)Application number : 07-041126
(22)Date of filing : 28.02.1995

(71)Applicant : NEC CORP
(72)Inventor : SURETSUSHIYU JIYAGANASAN
JIEEMUSU EFU FUIRUBIN

(30)Priority
Priority number : 94 221026 Priority date : 31.03.1994 Priority country : US

(54) CONTROL SYSTEM FOR HIGH-LEVEL PARALLEL COMPUTER SYSTEM USING SOFTWARE ARCHITECTURE

(57)Abstract:
PURPOSE: To provide the control system which uses the software architecture, equipped with several layers of abstract bodies, controlling the high-level parallel computer system.
CONSTITUTION: An abstract physical machine 10 (1st layer) includes a group of abstract physical processors and is considered as a microkernel. A 2nd layer includes a virtual machine 2 and virtual processors 16. The virtual machine is equipped with a virtual address space and a group of virtual processors connected by virtual topology. The virtual machine is mapped to the abstract physical machines and each virtual processor is mapped to the abstract physical processors. A 3rd layer defines threads 18. The threads are processes with light weight which run on the virtual processors. The abstract physical machine, abstract physical processors, virtual machines, virtual processors, thread groups, and threads are preferably all objects of a first class.



LEGAL STATUS

[Date of request for examination]	01.03.1995
[Date of sending the examiner's decision of rejection]	27.05.1998
[Kind of final disposal of application other than the examiner's decision of rejection or application converted registration]	
[Date of final disposal for application]	
[Patent number]	2848262
[Date of registration]	06.11.1998
[Number of appeal against examiner's decision of rejection]	10-10006
[Date of requesting appeal against examiner's	25.06.1998

decision of rejection]
[Date of extinction of right]

(19)日本国特許庁（ＪＰ）　(12) **公開特許公報（A）**　(11)特許出願公開番号

特開平7－271738

(43)公開日　平成７年(1995)10月20日

(51)Int.Cl.[6]	識別記号	庁内整理番号	Ｆ Ｉ	技術表示箇所
Ｇ０６Ｆ　15/16	４３０　Ｃ			
9/46	３５０	7737－5B		

審査請求　有　　請求項の数20　ＯＬ　（全 30 頁）

(21)出願番号　特願平7－41126

(22)出願日　平成７年(1995)２月28日

(31)優先権主張番号　０８／２２１０２６

(32)優先日　1994年３月31日

(33)優先権主張国　米国（ＵＳ）

(71)出願人　000004237
日本電気株式会社
東京都港区芝五丁目７番１号

(72)発明者　スレッシュ　ジャガナサン
アメリカ合衆国，95134　カリフォルニア，サン　ジョーズ，リオ　ロブルス　110　エヌ　イー　シー　アメリカ，インコーポレイテッド内

(72)発明者　ジェームス　エフ．フィルピン
アメリカ合衆国，95134　カリフォルニア，サン　ジョーズ，リオ　ロブルス　110　エヌ　イー　シー　アメリカ，インコーポレイテッド内

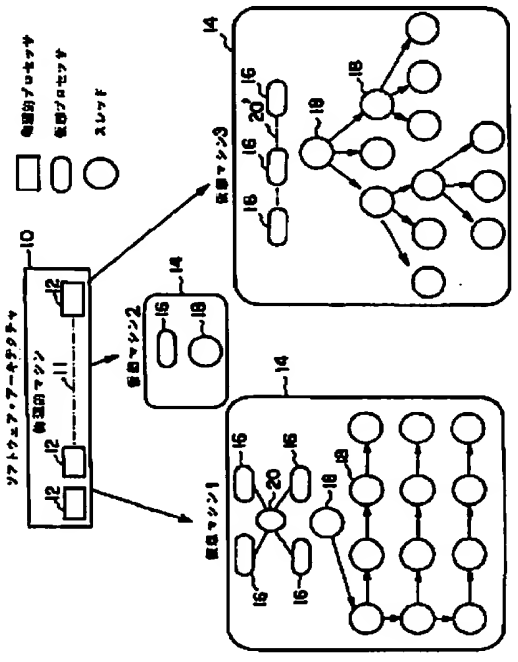
(74)代理人　弁理士　後藤　洋介　（外２名）

(54)【発明の名称】　ソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式

(57)【要約】

【目的】　いくつかの抽象体の層を備えた、高度並列コンピュータ・システムを制御するソフトウエア・アーキテクチャを用いた制御方式の提供。

【構成】　抽象物理的マシン１０（第１層）は抽象物理的プロセッサの組を含んでおり、マイクロカーネルと考えることができる。第２層は仮想マシン２と仮想プロセッサ１６とを含んでいる。仮想マシンは仮想アドレス空間と、仮想トポロジーで接続された仮想プロセッサの組とを備えている。仮想マシンは抽象物理的マシンにマッピングされ、各仮想プロセッサは抽象物理的プロセッサにマッピングされている。第３層は、スレッド１８を定義している。スレッドは、仮想プロセッサ上でランするライトウエイトのプロセスである。望ましくは、抽象物理的マシン、抽象物理的プロセッサ、仮想マシン、仮想プロセッサ、スレッド・グループ、ならびにスレッドはすべてファーアストクラスのオブジェクトである。



1

【特許請求の範囲】

【請求項1】　高度並列コンピュータ・システムを制御するためのソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式において、

一つのマイクロカーネルを形成する複数の抽象物理的プロセッサを備えた複数の抽象物理的マシンと；前記複数の抽象物理的プロセッサに付随し、複数の仮想プロセッサを備えた複数の仮想マシンと；前記複数の仮想プロセッサ上でランする複数のスレッドを備えた複数のスレッド・グループとを備え、

前記複数の仮想プロセッサおよび前記複数のスレッドはファーストクラスのオブジェクトであることを特徴とするソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式。

【請求項2】　前記複数の仮想プロセッサは仮想トポロジーにおいて接続されていることを特徴とする請求項1記載のソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式。

【請求項3】　前記マイクロカーネルのポリシーを管理するマイクロカーネル・ポリシー・マネージャはユーザがカスタマイズできることを特徴とする請求項1記載のソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式。

【請求項4】　前記複数の仮想プロセッサは、前記複数のスレッドのポリシーを管理する複数のスレッド・ポリシー・マネージャのうち、ユーザが、どのスレッド・ポリシー・マネージャをカスタマイズできるかを含むことを特徴とする請求項1記載のソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式。

【請求項5】　前記複数のスレッド、前記複数の仮想プロセッサ、ならびに前記複数の抽象物理的プロセッサは、機能的に連携し、仮想トポロジーを構築することを特徴とする請求項1記載のソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式。

【請求項6】　前記仮想トポロジーはユーザがカスタマイズできることを特徴とする請求項5記載のソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式。

【請求項7】　前記複数のスレッドは、それらのそれぞれの実行コンテクストから分離でき、実行コンテクストの遅延された割り当てを許すことを特徴とする請求項1記載のソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式。

【請求項8】　複数の多様な形態のポートをさらに備えたことを特徴とする請求項1記載のソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式。

【請求項9】　前記複数の多様な形態のポートはそれぞれファーストクラスのオブジェクトであることを特徴とする請求項8記載のソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式。

2

【請求項10】　前記複数のスレッドは、一般データと複合データとを含むメッセージを送ることを特徴とする請求項8記載のソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式。

【請求項11】　前記複数のスレッドは、それらのそれぞれのローカル・スタックおよびヒープを、他の複数のスレッドとは独立に、ガーベッジ・コレクトすることを特徴とする請求項1記載のソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式。

【請求項12】　前記複数のスレッド・グループはそれらのそれぞれの共有ヒープを、無関係の複数のスレッド・グループとは独立に、集めることを特徴とする請求項1記載のソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式。

【請求項13】　前記複数の仮想プロセッサは前記複数の抽象物理的プロセッサ上に多重化されていることを特徴とする請求項1記載のソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式。

【請求項14】　前記複数の仮想プロセッサ、前記複数の仮想マシン、ならびに前記複数のスレッドは、持続性メモリ内に存在することを特徴とする請求項1記載のソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式。

【請求項15】　前記複数の抽象物理的プロセッサはファーストクラスのオブジェクトであることを特徴とする請求項1記載のソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式。

【請求項16】　前記複数の仮想マシンはファーストクラスのオブジェクトであることを特徴とする請求項15記載のソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式。

【請求項17】　前記複数の抽象物理的マシンおよび前記複数のスレッド・グループはファーストクラスのオブジェクトであることを特徴とする請求項16記載のソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式。

【請求項18】　各々が仮想プロセッサ・コントローラと仮想プロセッサ・ポリシー・マネージャとを有し、物理的トポロジーにおいて接続された複数の抽象物理的プロセッサと；各々が、仮想アドレス空間と複数の仮想プロセッサとを有する複数の仮想マシンと；を備えたコンピュータ・システムであって、

前記複数の仮想マシンの各々の前記複数の仮想プロセッサは、前記仮想プロセッサ・コントローラ及び前記仮想プロセッサ・ポリシー・マネージャに応答して実行し、かつ、スレッド・コントローラとスレッド・ポリシー・

3

マネージャとを有し、前記複数の仮想プロセッサは仮想トポロジーにおいて接続され、各仮想プロセッサはそれぞれの抽象物理的プロセッサにマッピングされており、前記コンピュータ・システムは、前記スレッド・コントローラと前記スレッド・ポリシー・マネージャとに応答する前記複数の仮想プロセッサ上でランする複数のスレッドを、更に、備えていることを特徴とするコンピュータ・システム。

【請求項19】　前記複数の仮想プロセッサは前記複数の抽象物理的プロセッサ上で多重化されていることを特徴とする請求項18記載のコンピュータ・システム。

【請求項20】　更に、持続性メモリを備え、この持続性メモリには、前記複数のスレッド、前記複数の仮想プロセッサ、ならびに前記複数の仮想マシンを含むオブジェクトが存在することを特徴とする請求項18記載のコンピュータ・システム。

【発明の詳細な説明】

【0001】

【産業上の利用分野】本発明は、高度に並列化したコンピュータ・システムを制御するためのコンピュータ・ソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式に関し、特に、現代のプログラミング言語に対して、非常に効率的なサブストレートとして役立つよう設計したコンピュータ・ソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式に関するものである。

【0002】このコンピュータ・ソフトウエア・アーキテクチャを用いた制御方式は、制御の問題をポリシーの問題から分離したオペレーティング・システムにもとづいている。この分離はシステムの2つの異なる抽象体レベルで行っている。すなわち抽象物理的プロセッサと仮想プロセッサとにおいてである。これら抽象体のそれぞれは2つの構成要素に分れている。1つは、抽象体の制御部分を実現する“コントローラ”であり、もう1つはコントローラに対してポリシーを決定する“ポリシー・マネージャ”である。制御をポリシーから分離することによって、機能的に同一のシステムに対する異なる振舞の定義を、抽象体のポリシー・マネージャ部分を変更するのみで行うことが可能となる。

【0003】具体的には、このソフトウエア・アーキテクチャを用いた制御方式は、制御のライトウエイト・スレッドと仮想プロセッサとをファーストクラスのオブジェクトとしてサポートする。並行（コンカレンシー）マネジメントは、ファーストクラスのプロシージャおよびファーストクラスのコンティニュエーションによって実現する。それによって、ユーザが基本的なランタイム・システムに関する知識を持っていなくても、アプリケーションのランタイムの振舞を最適化することが可能となる。

【0004】さらに具体的には、本発明は、非同期並行

4

構造の構築と、基本的制御メカニズムとしてコンティニュエーションを用いてスレッド・コントローラの実現と、大規模並行計算の組織化と、並列計算のための強固なプログラミング環境との設計に関するものである。

【0005】

【従来の技術】並列計算に対する興味が高まり、その結果、並行性を表現するために高レベルのプログラムとデータの構造を明確に定義する並列プログラミング言語が多数、生み出された。非数値的アプリケーション領域をターゲットにする並列言語は典型的に、動的なライトウエイト・プロセスの生成、高レベル同期基本命令、分散データ構造、ならびにスペキュラティブな並行性を実現する並行構造をサポートする（効率は異なっている）。これらの並列言語は事実上すべて、2つの部分言語から成ると考えられる。2つの部分言語とは、すなわち、プロセスの集まりのアクティビティを管理し同期化する調整言語と、与えられたプロセスに限定されるデータ・オブジェクトを扱う計算言語とである。

【0006】伝統的に、オペレーティング・システムにはいくつかのクラスがある。例えば、リアルタイム、インタラクティブ（会話型）、バッチなどである。これら3つのクラスのオペレーティング・システムはユーザに対して異なるインターフェースを提供するので、プログラムをあるクラスのオペレーティング・システム（OS）から他のクラスのオペレーティング・システムに移動するのは困難であった。さらに、各クラスのオペレーティング・システムが決定するスケジューリングは異なっているので、1つのオペレーティング・システムのためのプログラム（例えばリアルタイム・アプリケーション）を他のオペレーティング・システム（例えば会話型開発システム）でデバッグすることは難しく、またアプリケーションがターゲット・システム上で正確かつ効率的にランすることに関して自信を持つことは困難である。

【0007】そして、これらのクラスのシステムではそれぞれ異なるスケジューリング方式を用いているので状況はさらに複雑である。例えば、ある種のリアルタイム・システムでは、複数のプロセスに対してスケジューリングの順序は固定しているのに対して、別のシステムでは優先規律を用いたり、ランニング・クオンタムを用い、さらに他のシステムでは、それらを組み合せている。会話型オペレーティング・システムあるいはバッチ・オペレーティング・システムはスケジューリングに関してかなりの数の選択肢を有している。

【0008】制御をポリシーから分離することによって、種々のクラスのオペレーティング・システムに対して容易にカスタマイズできるオペレーティング・システムを構築できる。本発明では、ポリシー・マネージャを実現するモジュールは典型的にはシステムのサイズに比べて非常に小さい。一般にコードのライン数は100未

5

満である。従って、ポリシーの振舞が異なるシステムを新たに構築する場合、通常はコードの小部分を書くのみでよい。また、ポリシー・マネージャは良く定義されたインターフェースを提供するので、ポリシーの振舞を変更した場合、システム全体を試験する必要はなく、新しいポリシー・マネージャだけを試験すればよい。

【0009】Hydra（参考文献："HYDRA／C.mmpi：An Experimental Computer System", William Wulf, Roy Lexia, 及びSamuel Harbison著, McGraw-Hill, 1991）は、制御とポリシーとの分離を意図して設計された最初のオペレーティング・システムである。しかし、Hydraはポリシーのカスタマイズをカーネルのレベルでしか認めていない。本発明ではさらに進めて、ポリシーの決定を、それらが特定のプログラムに関連するものである場合、カスタマイズできるようにする。従って、エディタやウインドウ・マネージャなどの会話型プログラムは、流体力学のシミュレーションや有限要素法の計算など、計算を主体とするプログラムとは非常に異なったポリシーを持つことができる。また、Hydraにおける制御とポリシーとの分離はコストのかかるものとなっている。それは、カーネルとポリシー・マネージャとの間に複数のコンテクスト・スイッチを必要とするからである。本発明では、ポリシー・マネージャは一般に適当なアドレス空間に直接リンクしており、コンテクストの切り換えは不要である。従って本発明のポリシー・マネージャは少なくとも従来のオペレーティング・システムにおけるポリシー・マネジメント（カスタマイズできない）と同程度に効率的であり、そして通常は従来以上に効率が良い。

【0010】高レベルの並列言語を実現する1つの方法は、専用の（ユーザ・レベル）仮想マシンを構築することである。仮想マシンは基本的に、調整部分言語に見られる高レベルの並行プリミティブを実現するサブストレートとして機能する。調整言語Lが並行プリミティブPをサポートする場合、Lの仮想マシン（L_r）の役割は、Pの実現に関連したことをすべて扱うことである。そのためにはマシンが、プロセスのスケジューリング、記憶、管理、同期化、その他を管理することがしばしば必要となる。しかし、L_rはPを効率良く実現するようにのみ調整されているので、非常に異なった並行プリミティブを実現することは多くの場合適当でない。従って、並行プリミティブP′によってLの方言を構築するためには通常、仮想マシンを新たに構築するか、あるいはPを用いてP′の意味規制を表現する必要がある。これら2つのアプローチには明らかに欠点がある。すなわち、第1のアプローチでは複雑な仮想マシンを新たに構築するため、コスト高である。一方、第2のアプローチは、Pの意味規制が高レベルであり、またL_rの機能が

6

限定されているので、不十分である。

【0011】言語の実現において、並行性を実現するために専用の仮想マシンを構築する代りに、低レベルのオペレーティング・システムの機能を用いることができる。プロセスの生成およびスケジューリングは、OSが管理する、制御のスレッド（ヘビーウエイトあるいはライトウエイト）によって実現する。そして同期化は、低レベルの、OSが管理する構造体を用いて扱う。このようにして実現したものは一般に、専用の実行時システムの周辺に構築したシステムより、ポータブルであり、また拡張性が高い。ただし、カーネル（低レベル）はすべて、アプリケーションとオペレーティング・システムとの間の保護境界を横断する必要があるので、効率は犠牲になる。さらに、汎用のOS機能は通常、対象の並行オペレータの意味規制に対して不感であるため、それらはコンパイル時間あるいは実行時間の点で最適化をほとんど、あるいはまったく行わない。

【0012】

【発明が解決しようとする課題】高度並列マルチプロセッサ／マルチコンピュータ・システムを制御するための、現代のプログラミング言語に対する非常に効率の良いサブストレートとして役立つコンピュータのオペレーティング・システム・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式が得られる。　更に、本発明によれば、カスタマイズ可能な仮想マシンにもとづく非同期の計算のためのソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式が得られる。

【0013】また、本発明によれば、仮想プロセッサ上でファーストクラスのオブジェクトとしてライトウエイト・スレッドをサポートするソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式が得られる。

【0014】更に、本発明によれば、カスタマイズ可能なポリシー・マネージャを、特にユーザ・レベルに含むソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式が得られる。

【0015】また、本発明によれば、カスタマイズ可能な仮想トポロジーを含むソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式が得られる。

【0016】また、本発明によれば、スレッド吸収、遅延TCB割り当て、ならびに記憶装置共有の場所としてのスレッド・グループを含むソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式が得られる。

【0017】更に、本発明によれば、多様な形態のポートを含むソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式が得られる。

【0018】また、本発明によれば、上述のようなソフ

7

トウエア・アーキテクチャを用いて制御されるコンピュータ・システムが得られる。

【0019】

【課題を解決するための手段】本発明によれば、高度並列コンピュータ・システムを制御するためのソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式において、一つのマイクロカーネルを形成する複数の抽象物理的プロセッサを備えた複数の抽象物理的マシンと；前記複数の抽象物理的プロセッサに付随し、複数の仮想プロセッサを備えた複数の仮想マシンと；前記複数の仮想プロセッサ上でランする複数のスレッドを備えた複数のスレッド・グループとを備え、前記複数の仮想プロセッサおよび前記複数のスレッドはファーストクラスのオブジェクトであることを特徴とするソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式が得られる。

【0020】更に本発明によれば、各々が仮想プロセッサ・コントローラと仮想プロセッサ・ポリシー・マネージャとを有し、物理的トポロジーにおいて接続された複数の抽象物理的プロセッサと；各々が、仮想アドレス空間と複数の仮想プロセッサとを有する複数の仮想マシンと；を備えたコンピュータ・システムであって、前記複数の仮想マシンの各々の前記複数の仮想プロセッサは、前記仮想プロセッサ・コントローラ及び前記仮想プロセッサ・ポリシー・マネージャに応答して実行し、かつ、スレッド・コントローラとスレッド・ポリシー・マネージャとを有し、前記複数の仮想プロセッサは仮想トポロジーにおいて接続され、各仮想プロセッサはそれぞれの抽象物理的プロセッサにマッピングされており、前記コンピュータ・システムは、前記スレッド・コントローラと前記スレッド・ポリシー・マネージャとに応答する前記複数の仮想プロセッサ上でランする複数のスレッドを、更に、備えていることを特徴とするコンピュータ・システムが得られる。

【0021】本発明は、高レベル・プログラミング言語のコンテクストにおいて広い範囲の並行構造体を表現することを可能とする調整サブストレートの実現に関するものである。本発明は汎用調整モデルを定義し、そのモデル上で、多数の特殊調整言語を効率良く実現できるようにする。本発明の実施においては、ソフトウエアのスキーム（Ｓｃｈｅｍｅ）（参考文献："Ｔｈｅ Ｒｅｖｉｓｅｄ Ｒｅｐｏｒｔ ｏｎ ｔｈｅ Ａｌｇｏｒｉｔｈｍｉｃ Ｌａｎｇｕａｇｅ Ｓｃｈｅｍｅ" ＡＣＭ Ｓｉｇｐｌａｎ Ｎｏｔｉｃｅｓ，２１（１２），１９８６，Ｊｏｎａｔｈａｎ Ｒｅｅｓ ａｎｄ Ｗｉｌｌｉａｍ Ｃｌｉｎｇｅｒ）を計算の基礎として用いた。スキームはより高次の、辞書的に見たときの、Ｌｉｓｐの方言である。スキームは望ましい言語ではあるが、当業者にとって明らかなように、上記調整サブストレートの設計は、いかなる現代の（高レベルの）プログラミング言語にも取り入れることができよう。

8

【0022】本発明のオペレーティング・システムは基本的に、共有メモリあるいは分離メモリを用いた、ＭＩＭＤ（マルチ・インストラクション－マルチ・データ）並列コンピュータ上でランするように設計され、またワークステーションのネットワークから成る分散マシン上でランするように設計されている。本発明のソフトウエア・アーキテクチャでは、分離メモリあるいは分散メモリを用いたマシン上で実行する場合には、共有仮想メモリ・モデルを用いる。その実現においては、異なる、並列のパラダイムに対応する多数の異なるアルゴリズムを用いた。上記並列には結果並列、マスター／スレーブ並列、ならびに論理的並列が含まれる。いくつかの異なる並列プログラミング・モデルを、フューチャー・ファーストクラスのタプル（組）空間、ならびにエンジンを備えたオペレーティング・システム上で実現した。

【0023】本発明の望ましい実施例のフィーチャーである、オペレーティング・システム（ＯＳ）を構成するスキームの方言（スティング（Ｓｔｉｎｇ）と呼ぶ）は、非同期、ライトウエイトの並行性を表現するための調整言語（専用仮想マシンによって実現）を含み、それは２つのアプローチの最良点を組み合せている。他の並列スキームのシステムおよび同種の高レベル言語の並列方言と異なり、スティングにおける基本的な並行オブジェクト（スレッド、仮想プロセッサ、ならびに物理的プロセッサ）は、ストリームライン化したデータ構造であり、複雑な同期化を必要としない。並行性の管理をＯＳによるサービスに依存する並列システムと違い、スティングはスキームのオブジェクトおよびプロシージャによってすべての並行管理の問題を実現し、その結果、ユーザは、背後のＯＳのサービスに関する知識を持つことなく、アプリケーションのランタイムの振舞を最適化することが可能となる。スティングは、さまざまな形態の非同期の並列性を生成し、管理するための基本的な特徴を、概念的に単一化したフレームワークで、かつ非常に一般的なフレームワークによってサポートする。結果として、高レベル言語の種々の並列方言をその上に構築できる効率的なサブストレートを構築できることが分った。スティングは単に、スタンドアロンの、短寿命のプログラムを実現する媒介手段とすることを意図したものではなく、並列計算のための豊かなプログラミング環境を構築するためのフレームワークを提供することを期待したものである。従って、このシステムは、スレッド・プリエンプション、スレッドごとの非同期のガーベッジ・コレクション、スレッド境界を越えた例外の扱い、ならびにアプリケーションに依存するスケジューリング・ポリシーをサポートする。さらに、このシステムは、持続性の長寿命なオブジェクト、マルチ・アドレス空間、その他、最新のプログラミング環境に共通する特徴を扱うために、必要な機能を有している。

9

【００２４】スティングでは、仮想プロセッサは抽象物理的プロセッサ上で多重化され、スレッドは仮想プロセッサ上で多重化される。この多重化に関連するポリシーの決定はすべて、ポリシー・マネージャによって行われる。物理的プロセッサ上の仮想プロセッサの多重化に関連する決定は、仮想プロセッサ・ポリシー・マネージャ（ＶＰＰＭ）によって行う、仮想プロセッサ上のスレッドの多重化に関する決定はスレッド・ポリシー・マネージャ（ＴＰＭ）によって行われる。

【００２５】ポリシー・マネージャは３つのタイプの決定を行う。すなわち、オブジェクトが生成あるいは再開されたとき、プロセッサ（物理的あるいは仮想）に新しいオブジェクト（ＶＰあるいはスレッド）をいかにマッピングするか、特定のプロセッサにマッピングされた複数のオブジェクトをランさせる順序、ならびにオブジェクトをあるプロセッサから他のプロセッサに、いつ再マッピングあるいは移動するかの３つである。

【００２６】スティングは、スキーム、スモールトーク（Ｓｍａｌｌ　Ｔａｌｋ）、ＭＬ、モジューラ３（Ｍｏｄｕｌａ３）、あるいはハスケル（Ｈａｓｋｅｌｌ）などの現代のプログラミング言語をサポートするように設計されたオペレーティング・システムである。スティングは、低レベルの直交構築体の基礎を与え、それによって言語の設計者あるいは使用者が、上記言語が必要とする種々の構築体を簡単かつ効率的に構築することを可能とする。

【００２７】現代のプログラミング言語は、従来のコボル、フォートラン、Ｃ、あるいはパスカルなどのプログラミング言語に比べ、より多くを要求する。スティングは現代のプログラミング言語をサポートするように設計されてはいるが、従来のプログラミング言語も同様に効率良くサポートする。現代のプログラミング言語が従来の言語と異なる点を以下にリストアップする。

【００２８】・並列性：汎用のマルチ・プロセッサはますます利用し易くなってきており、その結果、並行プログラミングのための効率的で、かつ表現力に優れたプラットフォームの構築に対して興味が高まっている。高レベルのプログラミング言語に並行性を組み入れるための努力は大部分が、特殊目的の基本命令を言語に付加するという点に払われている。

【００２９】・マルチ同期化モデル：並列プログラミングあるいは非同期プログラミングにおいて、多くの同期化プロトコルが用いられている。現代のオペレーティング環境は、できる限りさまざまなプロトコルをサポートする基本命令を提供するものでなければならない。

【００３０】・レイジー（遅延）評価およびイーガー評価：現代の多くの言語はレイジー評価あるいはイーガー評価のいずれか、または両方をサポートしている。オペレーティング・システムにとって、レイジーからイーガーまでの完全な評価ストラテジーを用意することは重要

10

である。

【００３１】・自動記憶管理：これは現代の多数の言語の基本的な特徴となっている。それは、自動記憶管理によってプログラムを一層、表現力に優れたものにでき、同時にプログラムのエラーを低減し、かつプログラムの複雑さを緩和できるからである。

【００３２】・トポロジー・マッピング：多くのプログラミング言語ではまだサポートされていないが、プログラムにおける通信オーバーヘッドを低減するように、処理のプロセッサへのマッピングを制御する能力は、マルチ・プロセッサ・コンピュータ・システムのサイズが大きくなり続け、かつトポロジーがより複雑になる以上、より重要なものとなろう。

【００３３】スティングはこれら種々の要素を効率良くサポートする。スティングは、現在利用できるものより一層、一般的でかつより効率的なアーキテクチャ・フレームワークにおいてこれを行う。スティングはまた、高い表現力および制御能力と、非並列レベルのカスタマイズ能を、プログラムに提供する。

【００３４】スティングは、その設計における４つの特徴によって、他の並列言語から最もよく区別できる。

【００３５】１．並行抽象体：並行性はスティングでは制御のライトウエイト・スレッドによって表現される。スレッドは非厳密な、ファーストクラスのデータ構造である。

【００３６】２．プロセッサ抽象体およびポリシー抽象体：スレッドは、スケジューリングおよび負荷平衡・プロトコルの抽象体を表す仮想プロセッサ（ＶＰ）上で実行する。仮想プロセッサの数は、実際に利用できる物理的プロセッサの数より多くてもかまわない。スレッドのように、仮想プロセッサはファーストクラスのオブジェクトである。１つのＶＰは１つのスレッド・ポリシー・マネージャを備え、このポリシー・マネージャはそれが実行するスレッドのためのスケジューリングと移行方式を決定する。異なるＶＰは、実際には、性能の低下無しに、異なるポリシー・マネージャを備えることができる。仮想プロセッサは、実際の物理的計算装置である物理的プロセッサ上で実行する。

【００３７】仮想プロセッサの集まりとアドレス空間とは組合わさって、１つの仮想マシンを形成する。複数の仮想マシンが単一の物理的マシン上で実行できる。物理的マシンは１組の物理的プロセッサから成る。仮想マシンおよび物理的マシンもまた指示可能な、スキームのオブジェクトであり、このオブジェクトとして操作可能である。

【００３８】３．記憶モデル：１つのスレッドはデータを、そのスレッドが排他的に管理するスタックおよびヒープに割り当てる。従って、複数のスレッドは、互いに独立にそれらのプライベート・ステートのガーベッジ・コレクションを行う。あるスレッドがプライベートのガ

11

ーベッジ・コレクションを始動する場合、グローバルな同期化は不要である。データはスレッドを横断して参照できる。スレッド境界を越えてオブジェクトのガーベッジ・コレクションを行うとき、領域間の参照情報が用いられる。記憶は世代スキャベンジング・コレクタによって管理される。１つのスレッドによって割り当てられた長寿命データあるいは持続データは、同じ仮想マシンにおける他のスレッドもアクセスできる。

【００３９】本発明の設計は記憶のローカリティということに配慮している。例えば、スレッドをランさせるための記憶装置はＶＰにキャッシュされ、そして１つのスレッドが終了したとき、すぐに再利用できるようリサイクルされる。さらに、複数のスレッドは、データの依存性が保証されるときは常に、同じ実行コンテクストを共有することができる。

【００４０】４．プログラム・モデル：スティングは、スレッド間で横断的に扱われるべき例外を許容し、ノン・ブロッキングＩ／Ｏをサポートし、仮想プロセッサのスケジューリングのカスタマイズを、仮想プロセッサ上のスレッドのスケジューリングがカスタマイズ可能であるのと同様に、可能とし、そしてマルチ・アドレス空間および共有持続オブジェクトを実現する内部構造を与える。スティングはまた、ファーストクラスの多様な形態のポートを用いたメッセージの効率の良い受け渡しをサポートする。ポートは、分離メモリ・プラットフォーム上の共有メモリの実現において、オーバーヘッドを緩和するのに役立つ。

【００４１】本発明の高度並列コンピュータ・システムを制御するソフトウエア・アーキテクチャでは、オペレーティング・システム（スティング）、基本言語、ならびにコンパイラを１つの抽象的マシンに統合する。スタート点はスキームなどの高レベルプログラミング言語である。このプログラミング言語は、スレッド、仮想プロセッサ、ならびにポリシー・マネージャを含む効率的な抽象体によって拡大されている。この優れたオペレーティング・システムは、データのローカリティにプレミアムを付けるという現在のアーキテクチャのトレンドを有効に利用したメカニズムを含んでいる。

【００４２】その結果、並列計算のための効率の良い調整構造体を構築するメカニズムが得られた。ライトウエイトのスレッドを用いることにより、進歩的なプログラミング環境の基礎が得られる。データのローカリティをサポートすることによって、効率的な非同期システムが得られる。

【００４３】このシステムの性能にとって中心的なことは仮想トポロジーの概念である。仮想トポロジーは、仮想プロセッサの集まりにおける関係を定める。ツリー、グラフ、ハイパーキューブ、ならびにメッシュとして構成されたプロセッサ・トポロジーはよく知られたその例である。仮想プロセッサは、スレッドが実行するスケジューリング、マイグレーション、ならびに負荷平衡のポリシーを定義する抽象体である。この仮想トポロジーは、複雑なスレッドとプロセッサのマッピング（物理的相互接続の低レベルの詳細を抽象する）を定める、単純で表現力に優れた高レベルのフレームワークを与えるよう意図されている。

12

【００４４】計算によって生成されたスレッドは、仮想トポロジー内のプロセッサに対して、そのトポロジーに関連したマッピング機能によってマッピングされる。ユーザはこれらのマッピング機能を定義することができる。仮想トポロジーを用いて特定のマルチプロセッサ・プラットフォーム上でシステムが実現されている場合、仮想トポロジー内の仮想プロセッサをプラットフォーム内の物理的プロセッサにマッピングするプロシージャを定義することが可能である。

【００４５】コードそれ自身は、それが物理的プロセッサあるいは物理的プロセッサの相互接続に対する参照を含んでいない限り、マシンとは独立している。スレッド・マッピングとローカリティに関するすべてのことは、プログラムが用いる、仮想トポロジーの仕様と、プログラム実行時のトポロジー内のノードの通過の仕方において抽象される。

【００４６】仮想トポロジーとプロセッサ・マッピングの利益は、効率性だけでなく、移植性という点にもあり、それによって並列アルゴリズムの実現を個別の物理的トポロジーごとに特殊化する必要がなくなる。スレッドをプロセッサに関連づけるマッピング・アルゴリズムは、仮想トポロジーの一部として細かく指定されるので、プログラマは、スレッドがどのように仮想プロセッサに対してマッピングされるべきかを正確に管理できる。ある計算において通信が必要となることが分かっている場合、これらのスレッドを特定の仮想プロセッサに明確に割り当てられるという能力によって、暗黙的なマッピング・ストラテジーの場合より優れた負荷平衡を行える。並列アルゴリズムによって定義される制御とデータフローのグラフの構造は、種々の形で用いることができる。スレッドの集まりが共通のデータを共有している場合には、これらのスレッドが実行する仮想プロセッサを同一の物理的プロセッサにマッピングするトポロジーを構築することが可能である。仮想プロセッサは物理的プロセッサ上で、スレッドが仮想プロセッサ上で多重化されるのと同じようにして多重化される。あるスレッドの集まりが重要な相互の通信を必要とする場合には、それらのスレッドを、仮想トポロジーにおいて互いに接近したプロセッサにマッピングするトポロジーを構築することができる。スレッドT_1が、他のスレッドT_2が発生する値に対してデータ依存性を有している場合、T_1とT_2とは同一の仮想プロセッサにマッピングすることが合理的である。プロセッサがほとんどビジー状態となるグラニュラリティの細かいプログラムでは、同一また

は近いプロセッサ上のデータ依存スレッドに対してスケジューリングを行える能力によって、スレッドのグラニュラリティを改善する機会が与えられる。最後に、適応ツリー・アルゴリズムなど、ある種のアルゴリズムは計算の進行につれて展開するというプロセス構造を有している。これらのアルゴリズムは、仮想プロセッサの動的生成が可能なトポロジー上において最も良く実行される。

【００４７】このソフトウエア・アーキテクチャの他の優れた面として、効率的な汎用のマルチ・スレッドのオペレーティング・システムおよびプログラム環境の実現における、コンティニュエーションおよびファーストクラスのプロシージャの役割がある。コンティニュエーションは、状態遷移の操作、例外の扱い、ならびに重要な記憶の最適化を実現するために用いられる。コンティニュエーションは、プログラム・ポイントの抽象体である。コンティニュエーションは、１つの引数を有するプロシージャによって表され、このプロシージャは、引数が示すプログラム・ポイントから実行すべき残りの計算を定義している。

【００４８】スティングの仮想アドレス空間は１組の領域によって構成されている。領域は、一時的にあるいは空間的に強いローカリティを示すデータを組織化するために用いられる。スティングはさまざまな領域をサポートする。すなわち、スレッド制御ブロック、スタック、スレッド・プライベート・ヒープ、スレッド共有ヒープなどである。データは、それらの意図された仕様および寿命にもとづいて領域に割り当てられ、従って異なる領域は、それらに関連した異なるガーベッジ・コレクタを備えることになる。

【００４９】例外と割り込みは常に、スレッド・レベルのコンテクスト・スイッチの場合のように、あるスレッドの実行コンテクストにおいて扱われる。例外ハンドラーは通常のスキームのプロシージャによって実現され、そして例外のディスパッチは基本的にコンティニュエーションの操作を含んでいる。

【００５０】スティングが、制御のライトウエイト・スレッドの生成および管理が可能なプログラミング・システムである限り、いくつかの特性を、他の高レベル言語のために開発されたスレッド・パッケージ・システムと共有している。これらのシステムもスレッドを明らかなデータタイプと見ており、また、さまざまな程度にプリエンプションをサポートし、そしてある限定されたケースでは、プログラマが特別のスケジュール管理を指定することを可能としている。これらのシステムでは、スレッドの抽象体が調整部分言語を定めている。

【００５１】しかし、スティングはいくつかの重要な点でこれらのシステムと異なっている。第１に、スティングが使用するスケジューリングとマイグレーションのプロトコルは完全にカスタマイズできる。異なるアプリケーションは、スレッド・マネージャあるいは仮想プロセッサの抽象体を変更することなく、異なるスケジューラをランさせることができる。このようなカスタマイズは仮想マシン自身の組織化に適用することができる。第２に、スティングによるデータのローカリティのサポート、記憶の最適化、ならびにスレッドの吸収によるプロセスの抑圧は他のシステムでは行えない。さらに、スレッドのオペレーションはすべてスレッドの仮想マシン内で直接実現される。スレッドのオペレーションの実行のために実施すべき、低レベルのカーネルに対するコンテクスト・スイッチは無い。スティングは、長寿命のアプリケーション、持続性のオブジェクト、ならびにマルチ・アドレス空間をサポートすることを意図した抽象的マシンにおいて構築される。スレッド・パッケージは、それらが（定義によって）完全なプログラム環境を定めていないので、これらの機能はまったく提供しない。

【００５２】スティングはシステム・プログラミング言語として設計されているので、低レベルの平行抽象体を提供する。アプリケーション・ライブラリは直接スレッド・オブジェクトを生成でき、そしてそれら自身のスケジューリングおよびスレッド・マイグレーション・ストラテジーを定めることができる。高レベルの平行構築体はスレッドを用いて実現できるが、しかし効率が保証されるなら、システムはユーザがスレッドのオペレーションを上述のように直接利用することを禁止するものではない。具体的には、同一のアプリケーションは、同一の実行時の環境において、異なる意味規制と異なる効率で、平行抽象体を定めることができる。

【００５３】ある点でスティングは、他の進歩的マルチ・スレッド・オペレーティング・システムに似ている。例えば、スティングは、コール・バック、ユーザが管理するオーバー・インタラプト、ならびにユーザ・レベルの操作としてのローカル・アドレス空間の管理に伴うノンブロッキングＩ／Ｏコールをサポートしている。スティングはユーザ・レベルの事柄とカーネル・レベルの事柄とを分けている。物理的プロセッサは（特権を与えられた）システムのオペレーション、および複数の仮想マシンに跨るオペレーションを扱う。仮想プロセッサはユーザ・レベルのスレッドおよびローカル・アドレス空間の機能をすべて実現する。しかし、スティングはスキームの拡張方言であるため、典型的なオペレーティング・システム環境では提供されない高レベルのプログラミング言語の機能性および表現性を提供する。

【００５４】スティングは、非同期プログラミング基本命令を構築し、そして新しい並列プログラミングのパラダイムを実験するためのプラットフォームである。さらに、その設計では、異なる平行性の手法を競走的に評価することが可能である。スキームは、意味規制が良く定義され、全体的に簡素であり、そして効率的であるため、このような実験を行うための特に豊かな環境を提供

15

する。しかし、スティングの設計はそれ自身言語に依存しない。従って、いかなる高レベルプログラミング言語にも極めて容易に組み込むことができよう。

【００５５】スティングは単に、興味深いと思われる各平行パラダイムおよび各平行プリミティブに対してフックを与えるものではない。そうではなく、広範囲の並列プログラミング構造体に共通の基本構造および機能に焦点を当てている。従って、ブロッキングの実現は論理的な計算をサポートするために容易に用いられる。スレッドの実行を抑止するために用いられるスレッド吸収の最適化は、フューチャーとタプル空間の同期化を実現するのに非常に適しており、そして最後に、カスタマイズ可能なポリシー・マネージャは、他のさまざまなパラダイムに対して公正で効率的なスケジューラを構築することを可能とする。

【００５６】

【実施例】次に本発明の実施例について図面を参照して説明する。

【００５７】図１に本発明の一実施例による高度並列コンピュータ・システムを制御するためのソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式のブロック図を示す。

【００５８】抽象物理的マシン（ＰＭ）１０は、物理的トポロジー（ＰＴ）１１で互いに接続された抽象物理的プロセッサ（ＰＰ）１２により構成されている。この抽象物理的マシンは１組の仮想マシン（ＶＭ）１４を実行させるために用いられる。それに対して、各仮想マシンは、仮想トポロジー（ＶＴ）２０，２０′で接続された１つ以上の仮想プロセッサ（ＶＰ）１６を備えている。スレッド（Ｔ）１８は、同じ仮想マシン内の１つ以上の仮想プロセッサ上で実行する。さらに、特定のスレッドは、同じ仮想マシン１４内の異なる仮想プロセッサ間で移行（マイグレート）できる。スレッド・ポリシー・マネージャ（ＴＰＭ）１９（図２，図３に示す）はスレッドのスケジューリングおよびスレッドの負荷平衡・ポリシーを制御する。異なる要素間の関係および各要素の詳細を以下に説明する。

【００５９】ソフトウエア・アーキテクチャ（オペレーティング・システム・アーキテクチャという場合もある）は、いくつかの抽象体の層の配列と考えることができる（図２）。第１の層は抽象物理的マシン１０を含み、このマシンは抽象物理的プロセッサ１２の組を含んでいる。この層は、現状のオペレーティング・システムにおいてマイクロ・カーネルと呼ばれているものに対応している。次の層は仮想マシン１４および仮想プロセッサ１６を含んでいる。仮想マシンは、仮想アドレス空間と、仮想トポロジーで接続された仮想プロセッサの組とを備えている。仮想マシンは抽象物理的マシンにマッピングされ、その際、各仮想プロセッサは抽象物理的プロセッサにマッピングされる。抽象体の第３の層はスレッド１８である。これらのスレッドは、仮想プロセッサ上でランするライトウエイトのプロセスである。

16

【００６０】仮想トポロジーは、例えば、メッシュ・トポロジーで物理的に接続された物理的プロセッサにマッピングされる仮想プロセッサのツリーである。仮想トポロジーによって、プログラマは、実施すべきアルゴリズムに適した（仮想）トポロジーでプログラムを表現することが可能となる。スティングは、仮想トポロジーから、ターゲット・マシンの実際の物理的トポロジーへの効率的なマッピングを提供する。また、仮想トポロジーによって、並列プログラムを、異なる物理的トポロジー間で容易に移すことが可能となる。

【００６１】スティングの調整部分言語の主な構成要素は、ライトウエイト・スレッドと仮想プロセッサである。スレッドは、ローカル記憶装置（すなわち、レジスタ、スタック、ならびにヒープ）、コード、ならびに関連する状態情報（すなわち、ステータス、優先順位、プリエンプション・ビット、ロックなど）を含む単純なデータ構造である。それらは独立した制御の場所を定義している。このシステムは、スレッドが含むコードに対して制約を課さない。有効なスキームの表現はすべて、独立したプロセスとして扱われる。

【００６２】図２，図３に示すように、各仮想プロセッサ（ＶＰ）１６はスレッド・コントローラ（ＴＣ）１７を含み、このコントローラはスレッド上およびスレッド・ポリシー・マネージャ（ＴＰＭ）１９上で状態遷移機能を実施する。そして、スレッド・ポリシー・マネージャはスレッドのスケジューリングと負荷平衡／移行ポリシーの両方を実施する。同じ仮想マシン内で各ＶＰはスレッド・コントローラを共有するが、異なるＶＰは異なるスレッド・ポリシー・マネージャを持つことができる。

【００６３】仮想プロセッサ１６は物理的プロセッサ１２上に、スレッド１８が仮想プロセッサ上に多重化されているのと同じようにして多重化されている。各物理的プロセッサは、マルチプロセッサ環境における計算エンジンに対応している。各物理的プロセッサＰＰに関連しているのは仮想プロセッサ・コントローラ１３および仮想プロセッサ・ポリシー・マネージャ１５である。仮想プロセッサ・コントローラは、プリエンプションによって、あるいは明示的なリクエストによって、仮想プロセッサ間でコンテクスト・スイッチを行う。仮想プロセッサ・ポリシー・マネージャは、物理的プロセッサＰＰ上で実行する仮想プロセッサ１６に対するスケジューリングの決定を扱う。例えば、仮想プロセッサは、その上でスレッドが実行していない場合、そして他のＶＰからスレッドを移転できない場合には、物理的プロセッサの制御を放棄することができる。物理的プロセッサは、システム内のいかなる仮想マシンの仮想プロセッサをもランさせることができる。

17

【００６４】仮想マシンは単一のアドレス空間２４を含み、関連する仮想プロセッサはそれを排他的にアクセスすることができる。仮想マシンは、グローバル記憶プール２６内のグローバルな情報（例えば、ライブラリ、ファイル・システムなど）を共有することができ、そしてグローバル共有オブジェクト２８（すなわち、グローバル・アドレス空間にあるオブジェクト）をそれらのローカル・アドレス空間にマッピングする。仮想マシンはまた、アドレス空間内のすべての活性オブジェクトをトレースするために用いられる活性オブジェクト・グラフ（すなわち、ルート環境３０）のルートを含んでいる。

【００６５】すべてのスティング・オブジェクト（スレッド、ＶＰ、仮想マシンを含む）は持続性メモリ内に存在する。このメモリは個別領域の集合として構成されている。オブジェクトは、世代コレクタを用いて領域内に集められたガーベッジである。１つのオブジェクトはそのアドレス空間内の他のオブジェクトをすべて参照することができる。最初、オブジェクトは短寿命のスレッド・ローカル領域に存在する。ガーベッジ・コレクションから生き残ったオブジェクトは世代階層において上位に移る。この機能はユーザにとっては全く明らかである。

【００６６】ファーストクラスのオブジェクトは、プロシージャに対して引数として渡したり、結果としてプロシージャから戻したり、あるいはデータ構造内に記憶できるオブジェクトのことである。本発明の抽象物理的マシンの望ましい実施例では、抽象物理的プロセッサ、仮想マシン、仮想プロセッサ、スレッドのグループ、ならびにスレッドはすべてファーストクラスのオブジェクトである。他の実施例では、スレッドおよび仮想プロセッサのみがファーストクラスのオブジェクトである。

【００６７】スティング・コンパイラはオービット（Ｏｒｂｉｔ）の改良バージョンである。オービットについてはD．Ｋｒａｎｚらの論文に記述されている（参考文献：“Ｏｒｂｉｔ：An Ｏｐｔｉｍｉｚｉｎｇ Ｃｏｍｐｉｌｅｒ ｆｏｒ Ｓｃｈｅｍｅ”，ｉｎ ＡＣＭ ＳＩＧＰＬＡＮ Ｎｏｔｉｃｅｓ，２１（７）：２１９－２３３，Ｊｕｌｙ １９８６）。コンパイラにより見えるターゲット・マシンは、現在ランしているスレッド・オブジェクトに対する参照を保持する専用のスレッド・レジスタを含んでいる。さらに、レジスタをコンテクスト・スイッチ上で退避、復元したり、あるいはスレッドの記憶領域（すなわち、スタックおよびヒープ）を割り当てたりするといった時間的な制約の厳しいオペレーションは、基本オペレーションとして用意される。連続するスキーム・プログラムは変更無しにコンパイルし、実行される。スティングでは、フューチャー、分散データ構造、ならびにスペキュラティブ平行オペレーションも実現している。スキーム・プログラムは、これらのパラダイムのいずれかによってサポートされた平行オペレーションによって自由に拡大させることができる。

18

【００６８】スレッドは、スティングにおけるファーストクラスのオブジェクトである。従って、それらは引数としてプロシージャに渡すことができ、また結果として戻し、さらにデータ構造内に格納することができる。スレッドは、スレッドを生成したオブジェクトより長く生き残ることができる。スレッドの状態は、サンク（ｔｈｕｎｋ）、すなわちスレッドが実行されるとき発動されるヌラリー（ｎｕｌｌａｒｙ）プロシージャを含んでいる。アプリケーションの値は終了時にスレッド内に格納される。例えば、（ｆｏｒｋ－ｔｈｒｅａｄ（＋ｙ（＊×ｚ））という表現を評価することによって、サンク（ｌａｍｂｄａ ()（＋ｙ（＊×ｚ））を発動する制御のライトウエイト・スレッドが生成される。このサンクの評価環境は、ｆｏｒｋ－ｔｈｒｅａｄという表現の辞書的環境である。

【００６９】スレッドは状態情報をその状態の一部として記録する（図４および図５参照）。スレッドは、遅延３６、スケジュール３８、評価４０、吸収４２、あるいは確定４４のいずれかの状態をとる。遅延されたスレッドは、スレッドの値が明確に要求されない限り、ランされることはない。スケジュールされたスレッドは、いずれかのＶＰが知っているスレッドであるが、まだ記憶資源は割り当てられていない。評価を行っているスレッドはランし始めたスレッドである。スレッドは、そのサンクのアプリケーションが結果を出すまでこの状態に留まる。上記結果が出たときスレッドの状態が確定する。吸収されたスレッドは、評価中のスレッドを特別化したものであり、重要であるため、以下にさらに詳しく説明する。

【００７０】状態情報および評価すべきコードに加えて、１つのスレッドはまた、（１）それが完了するのを待っている他のスレッドに対する参照情報と、（２）サンクの動的な、そして例外の環境と、スレッドの親、兄弟、ならびに子を含む系統情報とを含んでいる。

【００７１】各スレッドも、流体（すなわち動的）結合および例外の扱いを実現するために用いる動的な、そして例外の環境を有している。系統情報は、デバッグとプロファイリングのツールとして有用であり、それによってアプリケーションはプロセス・ツリーの動的な展開をモニタすることが可能となる。

【００７２】スレッドの実現においては、言語における他の基本オペレーションを変更する必要はない。スレッドの同期化意味規則は、例えばＭｕｌｔｉＬｉｓｐの“ｔｏｕｃｈ”や、Ｌｉｎｄａのタプル空間や、ＣＭＬの”ｓｙｎｃ”によって利用できる同期化機能をより一般的な（低レベルではあっても）形にしたものである。

【００７３】アプリケーションは状態を完全に制御し、その状態のもとで、ブロックされたスレッドを復活させることができる。しかし、データフロー（すなわちフューチャー・タッチ）、非決定論的な選択、ならびに制約

19

にもとづく同期化または障壁同期化に対する明示的なシステム・サポートがある。

【００７４】ユーザは、スレッド・コントローラ（ＴＣ）（スレッドのある状態において同期状態の遷移を実現する）が定義する１組のプロシージャ（以下にリストアップする）によってスレッドを操作する。ＴＣは、レジスタの退避および復元という２つの基本オペレーションを除いて、全体をスキームによって書くことが望ましい。スレッド・コントローラは記憶領域を割り当てない。従って、ＴＣのコールはガーベッジ・コレクションをトリガーしない。これらのオペレーションに加えて、スレッドは、プリエンプションのため、コントローラに入ることができる。スレッド・プロシージャを以下に示す。

【００７５】（ｆｏｒｋ－ｔｈｒｅａｄ ｅｘｐｒ ｖｐ）は、ｅｘｐｒを評価するためにスレッドを生成し、それをｖｐ上でランするようにスケジュールする。

【００７６】（ｄｅａｌｙ－ｔｈｒｅａｄ ｅｘｐｒ）は、（スレッド値によって）要求されたときｅｘｐｒを評価する遅延されたスレッドを生成する。

【００７７】（ｔｈｒｅａｄ－ｒｕｎ ｔｈｒｅａｄ ｖｐ）は、遅延された、ブロックされた、あるいは保留されたｔｈｒｅａｄをｖｐのレディー待ち行列に挿入する。

【００７８】（ｔｈｒｅａｄ－ｗａｉｔ ｔｈｒｅａｄ）は、このオペレーションを実行しているスレッドに、ｔｈｒｅａｄの状態が確定するまでブロックさせる。

【００７９】（ｔｈｒｅａｄ－ｂｌｏｃｋ ｔｈｒｅａｄ．ｂｌｏｃｋｅｒ）は、ｔｈｒｅａｄにブロックすることをリクエストする。ｂｌｏｃｋｅｒは、スレッドがブロックするときの条件である。

```
(let ((filter-list (list)))
  (sieve (lambda (thunk)
           (set filter-list (cons (dela
y-thread (thunk))
                                  (filt
er-list)))))))
```

では、フィルタがレイジーに生成される。フィルタは、一度要求されると、反復的に入力ストリームから要素を除去し、そして潜在的な素数を出力ストリーム上に発生する。ラウンド・ロビン・スレッド配置規律を用いるＶ※

```
(thread-run (car filer-list)
            (mod (1+ (vm. vp-vector (vp. v
m (current-virtual-processor))))
                 n))
```

（ｖｐ．ｖｍ（ｃｕｒｒｅｎｔ－ｖｉｒｔｕａｌ－ｐｒｏｃｃｓｓｏｒ））という表現は、現在のＶＰを一部とする仮想マシンを定義している。仮想マシンのパブリック・ステートは、その仮想プロセッサを収容するベクト

20

＊【００８０】（ｔｈｒｅａｄ－ｓｕｓｐｅｎｄ ｔｈｒｅａｄ．ｑｕａｎｔｕｍ）は、スレッドに実行の保留をリクエストする。ｑｕａｎｔｕｍ引数が与えられた場合には、指定された期間が経過したときスレッドは再開される。そうでない場合には、スレッドは、ｔｈｒｅａｄ－ｒｕｎを用いて明示的に再開されるまで、無期限に保留される。

【００８１】（ｔｈｒｅａｄ－ｔｅｒｍｉｎａｔｅ ｔｈｒｅａｄ．ｖａｌｕｅｓ）は、ｔｈｒｅａｄに対してｖａｌｕｅｓをその結果として終了することをリクエストする。（ｙｉｅｌｄ－ｐｒｏｃｅｓｓｏｒ）は、現在のスレッドに、そのＶＰの制御をやめるようリクエストする。このスレッドは適切なレディー待ち行列に挿入される。

【００８２】（ｃｕｒｒｅｎｔ－ｔｈｒｅａｄ）は、このオペレーションを実行しているスレッドを復帰する。

【００８３】（ｃｕｒｒｅｎｔ－ｖｉｒｔｕａｌ－ｐｒｏｃｃｓｓｏｒ）は、このオペレーションが、その上で評価されている仮想プロセッサを復帰される。

【００８４】ユーザがいかにスレッドをプログラムできるかを説明するため、図６のプログラムについて考える。このプログラムは、簡単な素数発見手段の実現を定義したものである。この定義ではいかなる特定の並行パラダイムも参照していない。このような問題はそのｏｐ引数によって抽象される。

【００８５】この素数発見手段の実現は、ストリーム・アクセスにおけるブロッキング・オペレーション（ｈｄ）、およびスレッドの最後に付加するアトミック・オペレーション（ａｔｔａｃｈ）を与える、ユーザが定義した同期スレッド抽象体に依存している。

【００８６】非同期の振舞の程度が異なる、素数発見手段の種々の処理を定義できる。例えば、

※Ｐ上でスケジュールした新しいフィルタを始動させるため、次のように書くことができる。

【００８７】

ルを含んでいる。

【００８８】シーブに対する上記コールを少し書き直すことにより、よりレイジーな素数発見手段の実現を表現できる。

【0089】

```
(let ((filter-list (list)))
  (sieve (lambdas (thunk)
           (set filter-list
             (cons (create-thread
                     (begin
                       (map threa
d-run filter list)
                       (thunk)))
               filter-list))
           (map thread-block filter
-list))
         n))
```

この定義では、潜在的な素数pに遭遇したフィルタは、レイジーなスレッド・オブジェクトLを生成し、チェーン内の他のすべてのフィルタにブロックすることをリクエストする。Lの値が要求されたときは、フィルタはチェーン内のすべての要素をアンロックし、そしてその入力スレッドにおけるpのすべての倍数を取り除く。この＊コールでは要求にもとづいて、シーブの拡張および入力の消費を抑制する。

【0090】このシーブは次のように、よりイーガーなバージョンに変えることもできる。

【0091】

```
(sieve (lambda (thunk)
         (fork-thread- (thunk) (curre
nt-vp)))
       n)
```

このアプリケーションを評価することによって、素数のすべての倍数を取り除くための新たなスレッドがスケジュールされる。このスレッドは、この操作を実行する仮想プロセッサ上でスケジュールされる。このコールでは、素数が新たに見つかるごとに、評価するスレッドが発生される。

【0092】スティングでは、スレッドのオペレーションを通常のプロシージャとして扱い、スレッドのオペレーションで参照されるオブジェクトを、スキームのどれか他のオブジェクトとして操作する。共通のストリームによって結ばれた2つのフィルタが終了した場合、上記ストリームが占有する記憶領域は再利用することができる。スティングは、スレッドのアクセスに対して先験的な同期化プロトコルを課さない。アプリケーション・プログラムが、スレッドの調整を整える抽象体を構築するようにしている。

【0093】フィルタによって生成されたスレッドは2つの方法の中の1つによって終了される。シーブに対するトップレベルのコールは、それがこれらのスレッドに対して明示的なハンドルを有するように、構成することができる。レイジーなシーブを生成するために用いるフィルタ・リスト・データ構造はその一例である。次に、

```
(map thread-terminate filter list)
```

を評価して、シーブ内のすべてのスレッドを終了させることができる。あるいは、アプリケーションはスレッドのグループを用いて、これらのスレッドを集合的に管理することができる。

【0094】＜スレッド・グループ＞スティングは、関連するスレッドの集まりに対する制御を獲得する手段としてスレッド・グループを与える。1つのスレッド・グループは、fork-thread-groupに対するコールによって生成される。このオペレーションは、新しいグループおよび新しいスレッドを生成し、新しいスレッドは新しいグループのルート・スレッドになる。子スレッドは、新しいグループを明示的に生成しない限り、同一のグループを、その親として共有する。1つのグループは1つの共有ヒープを含み、そのメンバーはすべてこのヒープをアクセスできる。スレッド・グループが次のコールによって終了したき、

```
(thread-group-terminate group)
```

グループ内の生きているスレッドはすべて終了され、その共有ヒープはガーベッジ・コレクトされる。

【0095】スレッド・グループはまた、そのメンバーに対して、それをすべてひとまとめにして適用できるデバッグ・オペレーションおよびスレッド・オペレーションも含んでいる。スレッド・グループは、デバッグおよびモニタのためのオペレーション（例えば、与えられたグループ内のすべてのスレッドのリストアップ、すべてのグループのリストアップ、プロファイリング、系統の報告など）と共に、通常のスレッドのオペレーション（例えば、終了、保留など）と同種のオペレーションを提供する。従って、スレッドTが終了したとき、ユーザ

はTのすべての子（終了されるべきTのグループの一部として定義されている）に対して次のようにリクエストできる。

【0096】（thread－group－terminate（thread．group T））

スレッド・グループは、階層的メモリ・アーキテクチャにおいて、共有を制御するための重要なツールである。グループのメンバーが共有するオブジェクトは、グループの共有ヒープ内に含まれているので、これらオブジェクトはメモリ内で物理的に互いに近接していることが望ましく、それによってより良いローカリティが得られる。スレッド・グループはまた、スケジューリングの場として用いることもできる。例えば、スレッド・ポリシー・マネージャは、グループ内のすべてのスレッドがランすることを許可されない限り、グループ内のスレッドはいずれもランできないというスケジューリング・ポリシーを実現できよう。このスケジューリング方式は“ギャング・スケジューリング”プロトコルと同種のものである。スレッド・グループはデータのローカリティを改善するために仮想トポロジーと共に用いることができる。

【0097】＜実行コンテクストおよびスレッド制御ブロック＞スレッドが評価を開始したとき、実行コンテクストがそれに対して割り当てられる。評価を行っているスレッドはいずれも、スレッド制御ブロック（TCB）32（図5）としても知られる実行コンテクストと関連している。TCBはコンティニュエーションを一般的に表したものであり、それ自身のスタック31とローカル・ヒープ33を含んでいる。スタックとヒープはともに拘束でき、そしてヒープは生成スキャベンジング・コレクタを用いてガーベッジ・コレクションされる。記憶オブジェクト以外に、TCBは関連するロックと、スレッドが最後にコンテクスト・スイッチを実行したとき残っている、生きたレジスタすべての値と、スレッドのサブステート（例えば、初期化、レディー、評価、ブロック、保留などの状態）と、スレッドが最後に実行されたVPと、スレッドの優先順位と、タイム・クオンタムとを含んでいる。

【0098】スレッド・ステートおよびスレッド・サブステートの遷移図を図4に示す。TCBの状態は、評価を行っているスレッド上で許可されたオペレーションを反映している。評価中のスレッドTがTCB T_{TCB} を有しているなら、T_{TCB} のステート・フィールドは以下の中のいずれか1つを示す。

【0099】初期化46：T_{TCB} に関連するスタックとヒープが初期化されているが、どのコードもまだ実行されていない。

【0100】レディー48：Tは利用できるいかなるVP上でも実行できるが、いずれのVP上でも現在、まだ実行されていない。

【0101】ラン50：TはあるVP上で現在実行されている。

【0102】ブロック52：Tは、あるスレッド上で、またはある条件のもとで現在ブロックされている。

【0103】保留54：Tは、基本的に無期限に保留されている。

【0104】終了56：Tは実行を終了し、残りの状態を一掃している。

【0105】スレッドとは異なり、TCBはファーストクラスの、ユーザに見えるオブジェクトではない。スレッド・コントローラとスレッド・ポリシー・マネージャのみがそれらをアクセスできる。新しいスレッドがランのレディー状態にあるとき、TCBはそれに割り当てられる。スレッドが確定状態となったとき、スレッド・コントローラはそのTCBを、後に生成されるスレッドのために、利用できる。TCBはユーザが維持するデータ構造内に逃げ込むことはない。TCBはシステム・レベルのプロシージャによって排他的に操作される。

【0106】スティングの実現はスレッドに対する記憶領域の割り当てを必要となるまで延期する。他のスレッド・パッケージでは、スレッドを生成する動作は、単にフォークされるべきスレッドに対する環境を設定するだけでなく、記憶領域の割り当ておよび初期化も含んでいる。このアプローチでは2つの重要な点で効率の低下を招く。第1に、グラニュラリティの細かい並列のもとでは、スレッド・コントローラは、実際にスレッドをランさせることより、それらを生成し、初期化することに、より長い時間を消費する。第2に、スタックおよびプロセス制御ブロックはスレッドが生成されると直ちに割り当てられるので、スレッド間のコンテクスト・スイッチはしばしば、キャッシュとページのローカリティの利点を活用できない。さらに、TCBの割り当てが遅延されない場合には、システムに必要な全メモリ容量は大幅に増加することになる。

【0107】スティングのスレッド制御ブロックは、仮想プロセッサによって管理されるリサイクル可能な資源である。TCBは、スレッドが評価を開始したときのみスレッドに対して割り当てられる。この割り当てのストラテジーはデータのローカリティを改善するように設計されている。TCBは、VP　V上でランするべきスレッドTに対して4つの方法の中の1つによって割り当てることができる。

【0108】1．現在V上で実行中のスレッドが終了した場合には、そのコンテクストは直ちに再割り当てのために利用できる。そのTCBは割り当てのための最も良い候補である。なぜなら、このTCBは、そのVPに対して最も高いローカリティを有しているからである。このVPに関連する物理的キャッシュおよびメモリは、最も最近VP上でランしたスレッドの実行コンテクストを含んでいる可能性が最も高い。

25

【０１０９】２．現在実行しているスレッドが終了していない場合には、Ｔに対するＴＣＢは、Ｖ上に維持されているＴＣＢのＬＩＦＯプールから割り当てられる。ここでも再び、上記実行コンテクストが、最も高いローカリティを有するものとなっている。

【０１１０】３．Ｖのプールが空の場合には、新しいＴＣＢが、これもＬＩＦＯの順序で構成されたグローバル・プールから割り当てられる。ローカルＶＰプールはいずれも、それが保持できるＴＣＢの数のしきい値τを維持している。プールがオーバーフローした場合には、そのＶＰは、ローカル・プール内のＴＣＢの半分をグローバル・プールに移動する。ローカル・プールがオーバーフローしていない場合には、τ／２ＴＣＢがグローバル・プールからＶＰのローカル・プールに移動される。グローバル・プールは２つの役割を果たす。すなわち、（１）ＴＣＢの割り当ておよび再使用におけるプログラムの振舞の影響を最小化すること、および（２）すべての仮想プロセッサに対するＴＣＢの公正な分配を保証することである。

【０１１１】４．最後に、ＴＣＢをグローバル・プールあるいはローカル・プールのいずれにおいても利用できない場合には、τ／２ＴＣＢの新しい組が動的に生成され、Ｔに割り当てられる。新しいＴＣＢは、グローバル・プールおよびＶＰのローカル・プールがともに空の場合にのみ生成されるので、スティング・プログラムの評価の際に実際に生成されるＴＣＢの数は、すべてのＶＰによって集合的に決められる。

【０１１２】＜仮想プロセッサ＞仮想プロセッサ（拡張して、仮想マシン）は、スティングではファーストクラスのオブジェクトである。ファーストクラスというＶＰの状態には、スティングを高レベルのスレッド・システムおよび他の非同期並列言語のいずれからも区別する重要な意味がある。第１に、明示的にプロセスを特定の仮想プロセッサにマッピングすることによって並列計算を組織できる。例えば、ＶＰ　Ｖ上で実行している他のスレッドＱと密接に通信することが知られているスレッドＰは、トポロジー的にＶに近いＶＰ上で実行すべきである。

26

＊る。スティングでは、ＶＰは直接的に示されるので、このような考慮を実現することができる。例えばシストリック・スタイルのプログラムは、現在のＶＰ（例えば、現在ＶＰ、左ＶＰ、右ＶＰ、上ＶＰなど）から離れて自己相対アドレシングを用いて表現することができる。このシステムは、多数の共通トポロジー（例えば、ハイパーキューブ、メッシュ、シストリック・アレーなど）に対していくつかのデフォールト・アドレシング・モードを提供する。さらに、ＶＰは特定の物理的プロセッサにマッピングできるので、ファーストクラスのデータ値として仮想プロセッサを操作できるという能力により、スティングのプログラムは、種々の特定のプロセッサ・トポロジーで定義される異なる並列アルゴリズムを極めて柔軟に表現することができる。

【０１１３】図７のプログラムを参照して説明する。このプログラムは、物理的プロセッサの２次元メッシュ上で多重化された仮想プロセッサの３次元メッシュを生成するものである。このアレーは、物理的マシンの高さおよび幅と同じ高さおよび幅を有している。深さ方向の各要素を同じ仮想プロセッサにマッピングすることによって、３次元アレーを２次元アレーに縮小する。従って、生成された仮想プロセッサの数は、物理的プロセッサの数と同じである。同じ深さのプロセッサにマッピングされたスレッドはすべて同じＶＰ上で実行する。プロシージャｇｅｔ－ｐｍ－ｈｅｉｇｈｔとｇｅｔ－ｐｍ－ｗｉｄｔｈは物理的マシン・インターフェースによって与えられる。仮想プロセッサの絶対アドレシングは、ｃｒｅａｔｅ－３Ｄ－ｍｅｓｈだけ戻したアレーへの単純なアレー参照である。

【０１１４】ｃｒｅａｔｅ－ｖｐプロシージャは、ｇｅｔ－ｐｐが戻した物理的プロセッサ上で走る新しいＶＰを生成する。トポロジーが生成されると、現在のＶＰから離れて自己相対アドレシング・プロシージャを構築することが可能である。例えば、トポロジーにおいて１ディメンジョン上方に移動する上ＶＰプロシージャを定義することができる。

【０１１５】

```
(define (up-VP)
  (let ((address (vp-address (currently-virtual-processor))))
    (array-ref 3D-mesh (vector-ref address 0))))
```

１．適当な物理的プロセッサにマッピングされる仮想プロセッサの組を生成する。

【０１１６】２．仮想トポロジーにおけるアドレスを各ＶＰに関連づける。

【０１１７】３．仮想トポロジーにおいて絶対アドレシングのために用いるデータ構造に仮想プロセッサを格納し、その構造上に適切なアクセスルーチンを定義する。

【０１１８】４．自己相対アドレシングのプロシージャを定義する。

【０１１９】＜スレッド・コントローラ＞スレッド・コントローラは、仮想プロセッサによる、物理的プロセッサやスレッドなど、他のシステム要素とのやり取りを扱う。スレッド・コントローラの最も重要な機能は、スレッドの状態遷移を扱うことである。スレッドが、その状態遷移によって、現在その上でランしている仮想プロセッサを生じた場合には、必ずスレッド・コントローラは

27

スレッド・ポリシー・マネージャをコールし、次にどのスレッドをランするべきかを決める。

【0120】スティングのスレッド・コントローラを実現する場合、いくつかの興味深い問題が明らかになる。中心的な状態遷移プロシージャは図9および図10に示す。これらのプロシージャで見られるTCBでの操作は、ユーザ・アプリケーションでは利用できない。スレッド・コントローラはスティングの中に書かれているので、TCプロシージャに対するすべての同期コールは通常のプロシージャ・コールとして扱われる。従って、現在のスレッドでランしているプロシージャが用いる活性レジスタは、コントローラへのエントリのとき、スレッドのTCB内に自動的に退避される。

【0121】プロシージャstart－context－switch（図8）は、その引数として、現在のスレッド（すなわち、TCに入ったスレッド）に対する望ましい次の状態をとる。プリエンプションは最初にディスエーブルされる。次に、新しいスレッド（あるいはTCB）が、プロシージャtpm－get－next－threadによって復帰される。

【0122】ランできるスレッドが無い場合には、プロシージャは偽（false）を戻す。この場合、現在のスレッドは再度ランされるか（レディー状態にあるとして）、あるいはプロシージャtpm－vp－idleが、現在のVPを引数としてコールされる。プロシージャtmp－vp－idleは種々の簿記操作を行うことができ、また、その物理的プロセッサに他のVPに切り換えるようリクエストすることができる。

【0123】次のオブジェクトが現在のTCBである場合、動作は一切行われず、現在のスレッドが直ちに再開される。戻されたオブジェクトが他のTCBの場合には、その状態がランに設定され、VPフィールドは現在のVPに設定される。そして、現在のTCBは（その状態がデッドの場合）TCBプール内でリサイクルされるか、またはそのレジスタが退避され、そして新しいTCBの状態が、プロセッサ・レジスタに復元される。

【0124】戻されたオブジェクトが、実行コンテクストを持たないスレッドの場合には、TCBがそれに対して割り当てられる。このTCBは、next－stateのフィールドがデッドの場合には現在のTCBとなる。あるいはVPローカル・プールまたはグローバル・プールから割り当てられるTCBとなる。スレッドは、基本プロシージャstart－new－tcbを用いて実行を開始する。このスレッドは、その実行コンテクストとして新しいTCBを用い、プロシージャstart－new－thread（図10参照）を応用する。

【0125】finish－context－switchのコード（図9）は、start－contcxt－switchが復帰させたスレッドによって実行される。その目的は、新しいスレッドのVPフィールドを設

28

定するためにスイッチ・アウトされたスレッド（このプロシージャ内で以前にコールされている）が保持するロックを解放し、適切であるなら以前のものをレディー待ち行列に組み入れ、プリエンプションタイムを再設定する。新しいスレッドがVP上に設定された後でのみ以前のものを待ち行列に組み入れることにより、コントローラは、状態遷移を起させることと、スレッドをVPのレディー待ち行列に組み入れることとの間の競合状態を排除する。プロシージャtmp－enqueue－ready－threadとtmp－enqueue－suspended－threadは、スレッド・ポリシー・マネージャによって実現される。

【0126】start－new－threadのコードを図10に示す。サンクE1を有するスレッド・オブジェクトは、それに対してTCBが割り当てられると、評価を開始でき、そしてデフォルト・エラー・ハンドラーおよび適当なクリンアップ・コードと関連するようになる。E1から出るためのスロー（start－new－threadが設定するキャッチポイント）はスレッド・スタックに適切に巻き戻させ、それによってスレッドが保持するロックなどの資源が適切に解放されるようにする。E1の評価に続く退出のコードはスレッドのスタックとヒープをガーベッジ・コレクションし、E1が生成した値をスレッド状態の一部として格納し、この値を待っているスレッドをすべて目覚めさせ、状態遷移プロシージャに対するテイル・リカーシブ・コールに、ランすべき新しいスレッドを選択させる。E1はダイナミック・ワインド・フォーム内に包まれているので、スレッドが異常終了した場合でも、スレッドの記憶領域がガーベッジ・コレクションされることが保証される。

【0127】ガーベッジ・コレクションは、スレッドのウエイターが起される前に行われなければならない。それは、スレッド（スレッドのサンクが復帰させたオブジェクトを含む）より長生きであって、ローカル・ヒープを含んでいたオブジェクトは他の活性ヒープに移転させる必要があるためである。これが行われないと、他のスレッドが、新たに終了したスレッドの記憶領域に対する参照を得ることになるからである。確定したスレッドの記憶領域は他のスレッドに割り当てられるので、これは明らかにエラーとなる。

【0128】＜スレッド・ポリシー・マネージャ＞各仮想プロセッサはスレッド・ポリシー・マネージャを有している。スレッド・ポリシー・マネージャは、仮想プロセッサ上でのスレッドのスケジューリングおよび移行に関するすべてのポリシーの決定を行う。スレッド・コントローラはスレッド・ポリシー・マネージャの依頼者であり、ユーザのコードはそれをアクセスできない。スレッド・コントローラは、次のことに関連して決定を行う必要がある場合には必ずスレッド・ポリシー・マネージャをコールする。すなわち、スレッドの仮想プロセッサ

29

への初期マッピングと、現在のスレッドがなんらかの理由で仮想プロセッサを解放したとき、次に仮想プロセッサはどのスレッドをランさせるべきかということと、いつ、どのスレッドを仮想プロセッサに、あるいは仮想プロセッサから移転させるかということである。

【０１２９】すべての仮想プロセッサは同一のスレッド・コントローラを有しているが、各仮想プロセッサは異なるポリシー・マネージャを備えることができる。このことは、各プロセッサが、必要なスケジューリングがさまざまに異なるサブシステムを制御するというリアルタイム・アプリケーションにとって特に重要である。

【０１３０】スレッド・ポリシー・マネージャはスレッド・コントローラに対してよく定義されたインターフェースを提供する。スレッド・ポリシー・マネージャが決定を行うために用いるデータ構造は、スレッド・ポリシー・マネージャにとって完全にプライベートなものとなっている。それらは特定のスレッド・ポリシー・マネージャに対してローカルとしたり、あるいは種々のスレッド・ポリシー・マネージャが共有するようにでき、また、それらの組み合せとすることもできる。しかし、システムの他の部分は一切利用できない。従って、スレッド・ポリシー・マネージャは、異なる仮想マシンに対して異なる振舞を行うようにカスタマイズすることができる。その結果、ユーザは、ランさせるプログラムの種類に応じてポリシーの決定をカスタマイズすることができる。

【０１３１】ＶＰはそれぞれ異なるスレッド・ポリシー・マネージャを備えることができるので、アプリケーションによって生成された異なるグループのスレッドは、異なるスケジューリング方式の対象とすることができる。仮想マシンあるいは仮想プロセッサは異なるスケジューリング・プロトコルあるいは異なるスケジューリング・ポリシーを扱うよう調整することができる。

【０１３２】スティングのスレッド・コントローラは、スレッドの状態遷移プロシージャを定義するが、先験的なスケジューリング・ポリシーあるいは先験的な負荷平衡・ポリシーは定義しない。これらのポリシーはアプリケーションに依存する場合がある。いくつかのデフォルト・ポリシーがスティングの全実行時間環境の一部として与えられるが、ユーザは自身のポリシーを自由に書くことができる。事実、図３に示すように、各仮想プロセッサ１６はそれ自身のスレッド・ポリシー・マネージャ（ＴＰＭ）１９を有している。従って、与えられた仮想マシン内の異なるＶＰは異なるポリシーを実現できる。ＴＰＭ１９はスレッドのスケジューリング、プロセッサ／スレッドのマッピング、ならびにスレッドの移行を扱う。

【０１３３】アプリケーションを個別のスケジューリング・グループに分けられるということは、長寿型の並列（あるいは会話型）プログラムにとって重要である。Ｉ／Ｏに関連したプロシージャを実行するスレッドは、計算に関連したルーチンを実行するスレッドとは異なるスケジューリングを必要とする。リアルタイムの制約を持つアプリケーションは、単純なＦＩＦＯスケジューリング・ポリシーのみを必要とするものとは異なるスケジューリング・プロトコルを用いて実現されるべきである。

30

【０１３４】ツリー構造の並列プログラムは、ＬＩＦＯにもとづくスケジューラを用いることによって、もっとも良好に動作しよう。マスタ／スレーブ・アルゴリズムあるいはワーカー・ファーム・アルゴリズムをランさせるアプリケーションは、公正さのためにラウンド・ロビン・プリエンプション・スケジューラを用いることによって、より良好に動作しよう。これらのアプリケーションはすべて、大きいプログラム構造体あるいは大きいプログラム環境の構成要素であるから、これらのアプリケーションを異なるポリシー・マネージャによって評価することで得られる柔軟性は重要である。同一の仮想マシン上で評価するスレッドの集まりを独立に実行する、個別のアプリケーションは存在し得る。さらに、各個別のスケジューラは、異なる性能特性を有し、そして異なる形で実現されたスレッド・ポリシー・マネージャを有することができる。

【０１３５】本発明は、柔軟なフレームワークの提供を探究するものである。そしてこの柔軟なフレームワークは、スレッド・コントローラ自身に対する変更を行うことなく、ユーザに対して明らかに異なるスケジューリング方式を組み入れることができるものである。そのため、すべてのＴＰＭは、その実現において制約は一切課されていないが、同一のインターフェースに従わなければならない。以下に示すインターフェースは、ランすべき新しいスレッドを選択し、評価中のスレッドを待ち行列に挿入し、スレッドの優先順位を設定し、そしてスレッドを移行させるためのオペレーションを提供する。これらのプロシージャはＴＣが排他的に用いるためのものである。一般に、ユーザ・アプリケーションは、スレッド・ポリシー・マネージャとスレッド・コントローラとのインターフェースを承知している必要はない。

【０１３６】（ｔｐｍ－ｇｅｔ－ｎｅｘｔ－ｔｈｒｅａｄ　ｖｐ）は次にｖｐ上でランすべきレディー状態のスレッドを戻す。

【０１３７】（ｔｐｍ－ｅｎｑｕｅｕｅ－ｒｅａｄｙ－ｔｈｒｅａｄ　ｖｐ　ｏｂｊ）は、スレッドあるいはＴＣＢのいずれかであろうｏｂｊをｖｐに関連するＴＰＭのレディー待ち行列に挿入する。

【０１３８】（ｔｐｍ－ｐｒｉｏｒｉｔｙ　ｐｒｉｏｒｉｔｙ）および（ｔｍｐ－ｑｕａｎｔｕｍ　ｑｕａｎｔｕｍ）は、それぞれの引数が有効な優先順位か、あるいは有効なクオンタムかを確認するガードプロシージャである。

【０１３９】（ｔｐｍ－ａｌｌｏｃａｔｅ－ｖｐ　ｔｈ

ｒｅａｄ）はｔｈｒｅａｄをｖｐに割り当てる。ｖｐが偽の場合には、ｔｈｒｅａｄはＴＰＭによって確定される仮想プロセッサに割り当てられる。

【０１４０】（ｔｍｐ－ｖｐ－ｉｄｌｅ　ｖｐ）は、ｖｐ上に評価を行っているスレッドが無い場合、スレッド・マネージャによってコールされる。このプロシージャはスレッドを他の仮想プロセッサから移行させたり、縛起を行ったり、他のＶＰに対するプロセッサ・スイッチ自身を持つために物理的プロセッサをコールしたりすることができる。

【０１４１】（ｔｐｍ－ｅｎｑｕｅｕｅ－ｓｕｓｐｅｎｄ　ｕｐ－ｔｈｒｅａｄ）は、ｖｐの保留待ち行列上のｔｈｒｅａｄを保留する。

【０１４２】ＴＰＭは、評価中のスレッドに対するスケジューリングの順序を決定する以外に、負荷平衡の２つの基本的決定を行う。（１）新しく生成されたスレッドを走らせるべきＶＰを選択する。（２）ＶＰ上のどのスレッドを移行できるかを決め、他のＶＰからどのスレッドを移行させるかを決める。

【０１４３】最初の決定は、初期の負荷平衡を扱うために重要である。第２の決定は、動的負荷平衡・プロトコルをサポートするために重要である。新しく評価中のスレッドの最初の配置の決定は、しばしば現在評価中のスレッドの移行を決めるために用いられる優先順位とは異なる優先順位にもとづいて行われる。ＴＰＭインターフェースはこの区別を保存する。

【０１４４】スケジューリング・ポリシーはいくつかの重要な事柄に従って分類できる。

【０１４５】ローカリティ：このシステム内に単一のグローバル待ち行列があるか、あるいは各ＴＰＭはそれ自身の待ち行列を持っているか？

状態：スレッドはそれらの現在の状態にもとづいて区別されているか・例えば、あるアプリケーションは、すべてのスレッドが、それらの現在の状態に関係無く単一の待ち行列を占めるという実現法を選択するかもしれない。あるいは、スレッドが評価中か、スケジュールされたか、保留されているかなどにもとづいて、スレッドを異なる待ち行列に分類することを選択するかもしれない。

【０１４６】順序付け：待ち行列は、ＦＩＦＯ、ＬＩＦＯ、ラウンド・ロビン、優先順位、あるいはリアルタイムの構造体として（他のものの中で）実現されているか？

直列化：アプリケーションはどのようなロッキング構造を種々のポリシー・マネージャの待ち行列に課すか。

【０１４７】この分類でどの選択肢を選ぶかによって、結果としての性能特性に差が生じる。例えば、評価中のスレッド（すなわち、ＴＣＢを有するスレッド）をスケジュールされたスレッドから区別するグラニュラリティ構造体を適合させ、そしてスケジュールされたスレッドのみを移行させることができるという制約を課した場合、評価中のスレッドの待ち行列をアクセスするのにロックは不要となる。この待ち行列は、それが生成されたＶＰに対してローカルである。しかし、スケジュールされたスレッドを保持している待ち行列は、他のＶＰ上のＴＰＭによる移行のターゲットであるから、ロックされなければならない。この種のスケジューリング方式は、動的負荷平衡が問題ではない場合には、有用である。従って、多数の長寿命の、非ブロッキング・スレッド（継続時間はほぼ同じ）が存在するときは、ほとんどのＶＰは、それら自身のローカル・レディー待ち行列上のスレッドの実行に、ほとんどの時間、ビジーとなる。従って、このようなアプリケーションにおけるこの待ち行列上のロックを除去することは有益である。一方、継続時間が変動するスレッドを発生するアプリケーションは、スケジュールされたスレッドおよび評価中のスレッドの両方の移行が可能なＴＰＭと共に用いたとき、ラン可能なレディー待ち行列をロックすることに伴ってコストがかかるが、より高いパフォーマンスを示す。

【０１４８】スレッド・ポリシー・マネージャが新しいスレッドを実行する必要があるときは常に、グローバル待ち行列はスレッド・ポリシー・マネージャ間の競合を意味する。しかし、このような仕組にすると、多くの並列アルゴリズムの実現において有用である。例えば、マスタ／スレーブ（あるいはワーカー・ファーム）プログラムでは、マスタに最初にスレッドのプールを生成する。これらのスレッドは、それら自身はいかなる新しいスレッドも生まない、長寿命の構造体である。これらは一度ＶＰ上でランすれば、滅多にブロックすることはない。従って、このようなスレッドを実行しているＴＰＭは、ローカル・スレッド待ち行列を維持することのオーバーヘッドをサポートする必要はない。しかし、ローカル待ち行列は、プロセスの構造がツリーあるいはグラフの形をとる結果、並列プログラムの実現においては有用である。これらの待ち行列は、仮想プロセッサの組において公正にスレッドをロード・バランスするために、このようなアプリケーションで用いることができる。

【０１４９】＜メッセージ伝達抽象体＞メッセージ伝達は分離メモリ・アーキテクチャにおいて効率の良い通信メカニズムでなければならない。特に、グラニュラリティの粗い並列アプリケーション、あるいは既知の通信パターンを有する並列アプリケーションに対してそうである。ポートは、分離メモリ・アーキテクチャ上で共有メモリを実現することのオーバーヘッドを最小限のものとするためにスティング内に設けられたデータ抽象体である。ファーストクラスのプロシージャおよびポートは、このコンテクストにおいて共同作業を示す。

【０１５０】スティングは、メッセージ伝達抽象体を共有メモリ環境において統合することを可能とする。ポートはファーストクラスのデータ・オブジェクトであり、

他のスレッドから送られるメッセージに対するレセプタクルとして働く。スティングは共有仮想メモリ・モデルを用いるので、いかなる複合データ構造（閉包を含む）でもポートを通じてやり取りできる。この柔軟性のため、スティングのアプリケーションはユーザ・レベルのメッセージ伝達プロトコルを明瞭な形で実現でき、そして単一化した環境において共有メモリとメッセージ伝達の最も優れた長所を結合することが可能となる。

【０１５１】ポートはファーストクラスのデータ構造である。ポートに対しては２つの基本的オペレーションがある。

【０１５２】１．（ｐｕｔ　ｏｂｊ　ｐｏｒｔ）は、ｏｂｊをｐｏｒｔにコピーする。この操作は送り手と非同期である。

【０１５３】２．（ｇｅｔ　ｐｏｒｔ）は、ｐｏｒｔ内の最初のメッセージを除去し、ｐｏｒｔが空の場合にはブロックする。

【０１５４】ポートＰから読み出したオブジェクトは、Ｐに書き込まれたオブジェクトのコピーである。このコピーは浅いコピーである。すなわち、オブジェクトの最上位の構造体のみがコピーされており、下位の構造体は＊

＊共有されている。これらのポートは、共有メモリが不十分な場合に用いるよう設計されているので、意味規則をコピーすることで設計されている。ｐｕｔの標準バージョンはシャローコピーを行うが、ディープコピーを行うバージョンもある。そのバージョンは、最上位のオブジェクトをコピーするだけでなく、下位の構造体もすべてコピーする。

【０１５５】例えば、浅いコピーを用いてメッセージ内の閉包を送る場合、閉包のコピーを構築する。しかし、閉包が定義する環境内で束ねられたオブジェクトへの参照は保存する。使用するコピー・メカニズムの選択は、明らかに背後の物理的アーキテクチャとアプリケーションの分野の影響を受ける。スティング実現が存在する特定の物理的サブストレートに適合させることのできる一連のメッセージ伝達実現が存在する。

【０１５６】従って、つぎの表現の評価により、

（ｐｕｔ（ｌａｍｂｄａ　()　Ｅ）ｐｏｒｔ）

プロシージャ（ｌａｍｂｄａ　()　Ｅ）の閉包がｐｏｒｔへ送出される。ｐｏｒｔ上でレシーバが次のように定義されているなら、

```
(define (receiver port)
  (let ((msg (get port)))
    (fork-thread (msg) (current-vp))
    (receiver)))
```

送出されたプロシージャはこのレシーバの仮想プロセッサ上で評価される。レシーバは、メッセージを評価するために新しいスレッドを生成することによって、古いリクエストの処理と並行して新しいリクエストを受け入れることができる。

【０１５７】このスタイルの通信は“アクティブ・メッセージ”と呼ばれている。それは、メッセージを受け取ったとき行うべき動作が、基本のシステムの一部としてコード化されておらず、メッセージそれ自身によって決められているからである。仮想プロセッサとスレッドのインターフェースは、メッセージ通信をサポートするためにいかなる変更も必要としないので、このようなモデルによって極めて大きい柔軟性と単純性が得られる。スティングの設計における２つのことが、この機能の実現にとって重要である。（１）オブジェクトが共有仮想メモリに存在するため、すべてのオブジェクト（他のオブジェクトに対するレファランスを有しているオブジェクト、例えば閉包を含む）は仮想プロセッサ間で自由に送信できる。（２）ファーストクラスのプロシージャは、ユーザが定義する複雑なメッセージ・ハンドラーの構築を可能とする。これらのハンドラーはいずれかの仮想プロセッサ上の分離したスレッド内で実行できる。分離メモリ・マシンでは、オブジェクトは分散共有仮想メモリに存在することになろう。説明のため、上述の例で、Ｅをデータベースの複雑な問い合わせとする。このデータベースが存在するプロセッサ上でレシーバが例示されたとすると、このような問い合わせは、データベース自身の、コストのかかる移行を伴わない。問い合わせは、データベースが存在するプロセッサに直接コピーされるので、通信のコストが低減される。データベースそれ自身は問い合わせを実行するプロセッサに移行する必要がない。より伝統的なＲＰＣスタイルの通信ではなくデータにプロシージャを送るという能力により、いくつかの点で重要なパフォーマンスおよび表現性の向上が得られる可能性がある。

【０１５８】ファーストクラスのプロシージャおよびライトウエイトのスレッドは、アクティブ・メッセージにおいて、魅力的な高レベルの通信抽象体を伝達する。これらの抽象体を利用せずにアクティブ・メッセージをサポートするシステムでは、この機能は典型的には低レベル・サポート・プロトコルによって実現される。ファーストクラスのプロシージャはアクティブ・メッセージを平凡に実現することを可能とする。アクティブ・メッセージはポートに送られるプロシージャである。ファーストクラスのポートは分散計算環境においても明確で重要な効用を有し、そして従来のＰＲＣより簡単で、かつ清潔なプログラミング・モデルの実現を可能とする。

【０１５９】＜メモリ管理＞スティングは共有仮想メモリ・モデルを用いる。分散メモリ・プラットフォーム上ではスティングは分散共有仮想メモリ・サブストレート

35

上で構築されなければならない。従って、参照の意味は、参照がどこで発生されているか、あるいはオブジェクトが物理的にどこにあるか、には依存しない。

【０１６０】＜記憶機構＞スティングでは各ＴＣＢ３２に関連して３つの記憶領域がある（図５）。第１はスタック３１であり、スレッドによって生成されたオブジェクトの割り当てに用いられる。このスレッドの寿命は、それを生成したものの動的な範囲を越えない。より正確には、スタック上に割り当てられたオブジェクトは、現在の（あるいは前の）スタック・フレームに割り当てられた他のオブジェクト、あるいはヒープに割り当てられた他のオブジェクトしか参照できない。スタックが割り当てられたオブジェクトはヒープ内のオブジェクトを参照することができる。なぜなら、そのスタックに関連するスレッドは、ヒープ３３がガーベッジ・コレクションされる間、保留となるからである。スタックに含まれている参照情報は、ガーベッジ・コレクタによってトレースしたとされるルートの一部である。

【０１６１】スレッドにとってプライベートなヒープ、すなわちローカル・ヒープ３３は、割り当てられた非共有オブジェクトに対して用いられる。このオブジェクトは、その寿命が、オブジェクトを生成したプロシージャの寿命を越える可能性がある。越える可能性があるとしたのは、スキームやＭＬなどのプログラミング言語ではコンパイラがオブジェクトの寿命を常に決めることができるとは限らないからである。さらに、未知のプロシージャに対するコールが可能な言語においては、オブジェクトの寿命が決められない場合もある。プライベート・ヒープに含まれている参照情報は同じプライベート・ヒープ内の他のオブジェクト、あるいは共有ヒープ、すなわちグローバル・ヒープ３５を示すことができるが、スタック３１内のオブジェクトを示すことはできない。このスタック内の参照情報はプライベート・ヒープ内のオブジェクトを示すことができるが、共有ヒープ内の参照情報はこれらのオブジェクトを示せない。プライベート・ヒープに割り当てられたデータは、単一の、制御のスレッドによって排他的に用いられるので、プライベート・ヒープによってより高いローカリティが実現する。複数のスレッド間にさしはさまれた割り当てがないということは、ヒープ内で互いに接近したオブジェクトは、論理的に互いに関連している可能性が高いことを意味する。

【０１６２】他のスレッドは、スレッドのスタックあるいはローカル・ヒープ内に含まれているオブジェクトをアクセスできない。従って、スレッドのスタックおよびローカル・ヒープは共に、同期化あるいはメモリのコヒーレンシーを考慮することなく、プロセッサ上のローカル・メモリにおいて実現することができる。スレッドのローカル・ヒープは実際には、世代的に組織した一連のヒープである。記憶領域の割り当ては常に、他の世代的

36

コレクタと同様に、最も若い世代において行われる。オブジェクトは、年齢が高くなるにつれて、古い世代に移動される。ローカル・ヒープのガーベッジ・コレクションはすべてスレッドそれ自身によって行われる。ガーベッジ・コレクションをサポートするほとんどのスレッド・システムでは、システム内のスレッドはすべて、ガーベッジ・コレクションの間は保留されなければならない。それに対して、スティングのスレッドは、他のスレッドと独立して、そして非同期的にそれらのローカル・ヒープをガーベッジ・コレクションする。従って、他のスレッドは、特定のスレッドがそのローカル・ヒープをコレクションしている間、計算を続けることができる。その結果、より優れた負荷平衡と高いスループットが得られる。このガーベッジ・コレクションのストラテジィの第２の長所は、ローカル・ヒープのガーベッジ・コレクションにかかるコストが、システム内のすべてのスレッドに課されるのではなく、記憶領域を割り当てるスレッドにのみ課されるという点にある。

【０１６３】スティングは、関連するスレッドの集まりを制御するための手段として“スレッド・グループ”を与える。子のスレッドは、それが新しいグループの一部として生成されたのでない限り、その親と同一のグループに属する。スレッド・グループは、デバッグおよびモニタのためのオペレーション（例えば、与えられたグループ内のすべてのスレッドのリストアップ、すべてのグループのリストアップ、プロファイリング、系統の報告など）と共に、通常のスレッドのオペレーション（例えば、終了、保留など）を与える。さらに、スレッド・グループはまた、そのメンバーがすべてアクセスできる“共有ヒープ”を含んでいる。

【０１６４】スレッド・グループの共有ヒープ、すなわちグローバル・ヒープ３５は、スレッド・グループが生成されたとき割り当てられる。ローカル・ヒープのような共有ヒープは実際には、世代的に組織された一連のヒープである。共有ヒープ内の参照情報は共有ヒープ内のオブジェクトしか示せない。これは、共有オブジェクトから参照されるオブジェクトはすべて共有オブジェクトであり、従って、共有ヒープ内に存在しなければならないからである。この共有ヒープに対する制約は、（ａ）共有ヒープ内にあるか、あるいは（ｂ）ローカル・ヒープ内に割り当てられていて、共有ヒープ内にガーベッジ・コレクションされているオブジェクトを、共有ヒープに記憶された参照情報が指示することを保証することによって、実施される。すなわち、参照されたオブジェクトから到達可能なオブジェクトのグラフは、共有ヒープ内にコピー、または配置されなければならない。このメモリ・モデルのオーバーヘッドは、ローカル・ヒープ上に割り当てられたオブジェクトに対する参照情報がどれくらい頻繁にエスケープするかによって決まる。経験によれば、ファイン・グレインド並列プログラムを実現す

る場合、ローカル・ヒープに割り当てられたオブジェクトはほとんど、関連するスレッドに対してローカルであり続け、共有されない。スレッド間で頻繁に共有されるオブジェクトは、言語抽象体あるいはコンパイル時の分析によって容易に検出される。

【０１６５】要約すると、あるスレッドに関連するスレッド領域間の参照規律は次のようになる。すなわち、（１）スタック内の参照情報は、その現在のあるいは以前のスタック・フレーム、またはローカル・ヒープ、または共有ヒープ内のオブジェクトを示し、（２）ローカル・ヒープ内の参照情報は、そのヒープ上のオブジェクトあるいはなんらかの共有ヒープに割り当てられたオブジェクトを示し、そして（３）共有ヒープ内の参照情報は、その共有ヒープ（あるいは、他のなんらかの共有ヒープ）に割り当てられたオブジェクトを示す。

【０１６６】ローカル・ヒープのように、グローバル・ヒープは世代的に組織されているが、グローバル・ヒープのガーベッジ・コレクションは、ローカル・ヒープに対するものより複雑である。それは、多数の異なるスレッドが、グローバル・ヒープ内のオブジェクトを同時にアクセスする場合があるからである。なお、その結果、グローバル・ヒープの割り当てにはヒープのロックが必要である。

【０１６７】グローバル・ヒープをガーベッジ・コレクションするために、関連するスレッド・グループ内のすべてのスレッド（そして、その下位のもの）は保留される。それは、これらのスレッドはすべてグローバル・ヒープ内のデータをアクセスできるからである。しかし、システム内の他のスレッド、すなわち、ガーベッジ・コレクションされるヒープに関連するグループ内にないものは、ガーベッジ・コレクションと無関係に実行を継続する。

【０１６８】各グローバル・ヒープは、それに関連し、そこに到来する参照情報を有している。これらの組は、領域境界を横断する参照情報の記憶に対するチェックによって、維持される。グローバル・ヒープに関連するスレッドが保留された後、ガーベッジ・コレクタは到来参照情報の組をガーベッジ・コレクションのためのルートとして用いる。到来参照情報の組から到達できるオブジェクトはすべて新しいヒープにコピーされる。ガーベッジ・コレクションが終了すると、グローバル・ヒープに関連したスレッドは再開される。

【０１６９】＜抽象物理的マシンおよび抽象物理的プロセッサ＞このオペレーティング・システムの最も低レベルの抽象体は、抽象物理的マシン（ＡＰＭ）と呼ばれるマイクロ・カーネルである。

【０１７０】ＡＰＭはスティング・ソフトウエア・アーキテクチャにおいて３つの重要な役割を果たす。

【０１７１】１．複数の仮想マシンをサポートする安全で効率的な基礎を提供する。

【０１７２】２．システム内の他のすべての要素を、ハードウエアに依存する特徴および特異性から分離する。

【０１７３】３．システムの物理的ハードウエアに対するアクセスを制御する。

【０１７４】ＡＰＭはルート仮想マシンと呼ばれる特別の仮想マシン内で実現される。このマシンは、仮想アドレス空間、仮想プロセッサ、ならびにスレッドを含む、他のいずれの仮想マシンでも利用できる機能に対するアクセス手段を有している。さらに、ルート仮想マシンは、抽象物理的プロセッサ、デバイス・ドライバ、ならびに仮想メモリ・マネージャに対するアクセス手段を有している。抽象物理的マシンは仮想マシンによって構成されており、その結果、いくつかの重要な表現性が得られる。ヘビーウエイトのスレッドは一切無い。すべてのスレッドはライトウエイトである。システム・コールを実現するカーネル・スレッドあるいはスタックは無い。すべてのシステム・コールは、システム・コールを作成するスレッドの実行コンテクストを用いて扱われる。このことは、スキームが安全な言語であり（すなわち、ダングリング・ポインタ、アドレスとデータ間の自由強制などは不可能である）、そしてＡＰＭの部分はシステム内のすべての仮想マシンにマッピングされているため、可能となっている。ユーザのスレッドが利用できる非同期のプログラミング構築体は、ＡＰＭ内のスレッドも利用できる。ＡＰＭに関連したスレッドは、仮想マシン内の他のすべてのスレッドと同様に制御することができる。カーネル操作の実行をブロックするスレッドは、そのことを、それらスレッドの仮想プロセッサに通知する。それによってＶＰは他のなんらかのスレッドを自由に実行できる。これはスレッド間の通信およびＩ／Ｏの両方の場合に行われる。スティングは、例えば、スケジューラの起動、あるいはＰｓｙｃｈｅの仮想プロセッサ抽象体と同じ能力を提供するように、非ブロッキング・カーネル・コールを処理する。

【０１７５】仮想マシンはＡＰＭによって生成され、そして破壊される。新しい仮想マシンの生成には以下のことが伴う。

【０１７６】１．新しい仮想アドレス空間を生成する。

【０１７７】２．このアドレス空間にＡＰＭカーネルをマッピングする。

【０１７８】３．この仮想マッピング内にルート仮想プロセッサを生成する。

【０１７９】４．このマッピングに抽象物理的プロセッサを割り当てる。

【０１８０】５．抽象物理的プロセッサ上でランさせるために上記ルート仮想プロセッサをスケジュールする。

【０１８１】仮想マシンの破壊には、そのマシン上でランしているすべてのスレッドを終了させるための信号を発生し、マシン内で実行しているスレッドがオープンしたデバイスをすべてクローズし、そして最後に、このマ

39

シンに関連する仮想アドレス空間の割り当てを解除することが伴う。

【０１８２】各プロセッサ抽象体１２は仮想プロセッサ・コントローラ（ＶＰＣ）１３と仮想プロセッサ・ポリシー・マネージャ（ＶＰＰＭ）１５から成る。ＶＰコントローラとＶＰポリシー・マネージャとの関係は、スレッド・コントローラとスレッド・ポリシー・マネージャとの関係と同種である、すなわちＶＰコントローラはＶＰポリシー・マネージャの依頼者である。ＶＰコントローラがポリシーの決定を行うことが必要となった場合には必ず、ＶＰコントローラはその決定を行うためにＶＰポリシー・マネージャをコールする。

【０１８３】物理的プロセッサはすべて同一のＶＰコントローラをランさせるが、それらは異なるＶＰポリシー・マネージャをランさせることができる。その結果、マルチプロセッサ・システムはシステムによる各物理的プロセッサの利用をカスタマイズすることが可能となる。また、システムは各物理的プロセッサ上で同じＶＰポリシー・マネージャをランさせることも可能である。

【０１８４】仮想マシンが抽象物理的プロセッサ上の仮想プロセッサをスケジュールしようとする場合、仮想マシンはその物理的プロセッサ上の仮想プロセッサ・コントローラをコールする。同様に、仮想マシンが、抽象物理的プロセッサから仮想プロセッサを除去しようとする場合には、仮想マシンはその物理的プロセッサ上の仮想プロセッサ・コントローラをコールする。各ＶＰコントローラは、仮想プロセッサの状態変化を含め、その物理的プロセッサにマッピングされた仮想プロセッサを管理する。

【０１８５】ＶＰポリシー・マネージャは、物理的プロセッサ上の仮想プロセッサのスケジューリングおよび移行に係わるすべてのポリシーの決定を行う。この決定には３つのタイプがある。第１に、ＶＰポリシー・マネージャはＶＰからＰＰへのマッピングを決める。このマッピングは２つの異なるタイミングで行われる。すなわち、ＶＰが最初にランされたときと、ブロックされていたＶＰが再びランされたときである。第２に、ポリシー・マネージャは、ＰＰ上のＶＰをランさせる順番と期間を決定する。最後に、ＶＰポリシー・マネージャは、いつＶＰをあるプロセッサから他のプロセッサに移動（移行）させるべきかを決める。

【０１８６】これらの３つの決定によって、ＶＰポリシー・マネージャはマシン上のワーク・ロードのバランスをとることができ、そして仮想マシンに関する物理的マシンの公正さに係わる性質を決めることができる。また、物理的プロセッサが故障したとき、故障許容ＶＭのＶＰをどこに移動させるかを決めることができる。

【０１８７】スレッド・ポリシー・マネージャのようにＶＰはＶＰコントローラに対して良く定義されたインターフェースを提供する。ＶＰポリシー・マネージャがその決定を行うために用いるデータ構造はＶＰポリシー・マネージャに対して完全にプライベートである。これらのデータ構造は特定のＶＰポリシー・マネージャに対してローカルとできるか、またはＶＰポリシー・マネージャの種々の場合において共有できるか、またはそれらの組み合せとできる。しかし、システムの他の要素はそれらに対するアクセス手段を持たない。ＶＰポリシー・マネージャは、スティングの異なる場合に対して異なる振舞をするようにカスタマイズすることができる。この機能により、スティングを、リアルタイム・システムや、会話型システムや、多量の計算を行うシステムなど、さまざまなオペレーティング・システム環境に対して、カスタマイズすることが可能となる。

40

【０１８８】最後に、スレッド・ポリシー・マネージャはスレッド間の負荷平衡および公正さに係わっているが、仮想プロセッサ・ポリシー・マネージャは、仮想マシン間および仮想プロセッサ間の負荷平衡および公正さに係わっている。

【０１８９】ＡＰＭ内の各物理的プロセッサは、仮想プロセッサ・コントローラ（ＶＰＣ）と仮想プロセッサ・ポリシー・マネージャ（ＶＰＰＭ）を含んでいる。この点で、物理的プロセッサは構造的に仮想プロセッサと同一である。ＶＰＣは仮想プロセッサ上の状態変化に影響を与える。スレッドのように、仮想プロセッサはラン、レディー、ブロック、あるいは終了のいずれかの状態をとり得る。ラン状態のＶＰは物理的プロセッサ上で現在実行されている。レディー状態のＶＰはランすることが可能であるが、現在はランしていない。ブロック状態のＶＰは、なんらかの外部イベント（例えばＩ／Ｏ）を待っているスレッドを実行している。ＶＰＰＭは物理的プロセッサ上のＶＰのスケジューリングを行う。そのスケジューリング・ポリシーはＴＰＭが用いるものと同様である。ＶＰＰＭは良く定義されたインターフェースをＶＰコントローラに対して提供する。異なるスティングのシステムは異なるＶＰポリシー・マネージャを備えることができる。

【０１９０】＜例外の扱い＞同期した例外および割込はスティングでは一様に扱われる。すべての例外には、例外を扱うための１組の動作を実行するハンドラーが関連している。ハンドラーはスレッド内で実行するプロシージャである。プロセッサＰ上で生じた例外は、Ｐの現在のスレッドのコンテクストを用いて実行する。スティングのマイクロ・カーネル内には特別の例外スタックは無い。プロセッサＰ上である例外（例えば、無効命令、メモリ保護破壊など）が生じた場合、Ｐの現在のコンティニュエーション（すなわち、プログラム・カウンタ、ヒープ・フロンティア、スタックなど）がまず退避される。次に例外ディスパッチャーは例外のターゲットを見つけるため、スレッドがランしている場合にはそれを中断し、そしてハンドラーのコンティニュエーションおよ

41

び引数をターゲット・スレッドのスタック上にプッシュする。次に、ディスパッチャーは（a）現在のスレッドを、単純にそれに復帰させることによって再開させるか、（b）ターゲット・スレッドを再開させるか、あるいは（c）このプロセッサ上の他のいずれかのスレッドを再開させるためにスレッド・コントローラをコールするか、いずれかを選択する。ターゲット・スレッドが再開された場合には、そのスレッドはそのスタックの最も上のコンティニュエーションを実行する。これは例外ハンドラーのコンティニュエーションである。

【0191】スティングにおけるこの例外処理手段はいくつかの点で優れている。

【0192】1．この例外処理手段はプロシージャであるため、単にそれをコールするだけで例外を扱える。

【0193】2．例外は、実行コンテクストを受け取るスレッドの実行コンテクストにおいて扱われる。

【0194】3．例外は現在のスレッドのコンテクストにおいてディスパッチされる。

【0195】4．一度ディスパッチされた例外はターゲット・スレッドの現在のコンティニュエーションとなり、そしてスレッドが再開されたとき自動的に実行される。

【0196】5．例外はターゲット・スレッドが再開されたときのみ扱われる。

42

*【0197】6．例外を扱うコードはスキームによって書かれ、そしてそのコードはコンティニュエーションとプロシージャを操作して所望の効果を達成する。

【0198】ファーストクラスのプロシージャとスレッド、明白なコンティニュエーション、動的な記憶領域の割り当て、ならびに均一なアドレシング・メカニズムはすべてスティングの設計の中心的な特徴であり、その結果、スティングはこの例外のモデルを与えることが可能となっている。

【0199】同期した例外のターゲット・スレッドは常に現在のスレッドである。非同期の例外、すなわち割り込みはわずかに異なる形で扱われる。割り込みはどのスレッド（現在実行中のスレッドではない）でも制御できるので、このような例外を扱うためには、ハンドラーは、例外を直接処理するか、すなわち現在ランしているスレッドを中断して例外を扱うか、あるいは新しいハンドラーを生成する必要がある。割込ハンドラーもスキームのプロシージャであるため、ハンドラーを実行するためにスレッドを確立するか、あるいは現在のスレッドを用いる場合、単に適当なスレッドの現在のコンティニュエーションを、ハンドラーをコールするように設定すればよい。スティングの例外ディスパッチャーのための疑似コードを以下に示す。

*【0200】

```
 1: (define (exception-dispatcher type.
args)
 2:    (save-current-continuation)
 3:    (let ((target handler (get-target
&handler type args)))
 4:    (cond ((eq? target (current-thread
)))
 5:           (apply handler args))
 6:          (else
 7:            (signal target handler a
rgs)
 8:            (case ((exception-priorit
y type))
 9:               ((continue) (return))
 10:              ((immediate) (switch-to
-thread target))
 11;              ((reschedule) (yield-pr
ocessor))))))))
```

ライン2では、現在のコンティニュエーションが現在のスレッドのスタックに退避される。このコンティニュエーションは、エスケープできず、そして一度だけコールされるので、上記スタックに上記コンティニュエーションを退避できる。ライン3では、ディスパッチャーが、例外の対象となるスレッドと、例外のタイプに対するハンドラーとを見つける。ライン4では、例外のターゲットが現在のスレッドであるかどうかがチェックされ、そうなら、例外コンティニュエーションはプッシュされない（ライン5）。ディスパッチャーはハンドラーをむしろ単純にその引数に適用する。ディスパッチャーはすでに例外ターゲット（すなわち現在のスレッド）のコンテクストで走っているので、このことが有効である。例外のターゲットが現在のスレッドでない場合には、ディスパッチャーは例外をターゲット・スレッドに送る（ライン7）。スレッドに信号を送ることはスレッドを中断

し、信号ハンドラーとその引数とを含むコンティニュエーションをスレッドのスタックにプッシュすること、そして信号ハンドラーが実行されるようにするスレッドを再開させることと等価である。ターゲット・スレッドに信号を送った後、ハンドラーはプロセッサ上で次にどのスレッドを走らせるかを決める（ライン８）。走らせるのはそれ自身の場合もあり（ライン９）、あるいはターゲット・スレッド（ライン１０）か、または最も優先順位の高いスレッドの場合もある（ライン１１）。

【０２０１】スティングの例外ハンドリング機能と他のオペレーティング・システムにおけるものとは、もう１つ重要な点で異なっている。例外を扱うスレッドは、システム内のユーザ・レベルのスレッドと違わないので（例えば、それらは自身のスタックとヒープを持っている）、また、例外ハンドラーは通常のファーストクラスのプロシージャであるため、ハンドラーは記憶領域を自由に割り当てることができる。ハンドラーによって生成されたデータは、他のデータが復元されるのと同じ方法で、ガーベッジ・コレクタによってリクレームされよう。例外ハンドリングのメカニズムと、より高レベルのスティングの抽象体との間の均一性のため、デバイス・ドライバを実現したとき、高い表現性および高い効率が得られる。このことは、上記均一性が無い場合には、並列言語あるいは並列オペレーティング・システムにおいて実現しない。

【０２０２】ファーストクラスのプロシージャとスレッド、明白なコンティニュエーション、動的な記憶領域の割り当て、ならびに均一なアドレシング・メカニズムがすべてスティングの設計の特徴であるため、スティングはこの例外のモデルを与えることができる。

【０２０３】＜並行パラダイム＞以上、ソフトウエア・アーキテクチャについて詳しく説明したが、以下においてはいくつかの広範は並行パラダイムについて説明し、本発明のソフトウエア・アーキテクチャによってそれを実現する。

【０２０４】結果としての並行プログラムでは、並行して実行する各プロセスは、複合データ構造（例えば、アレーあるいはリスト）の値に影響を与える。または各プロセスは複合プロセスのグラフのメンバーである。プロセスの通信はこの結果の構造体またはグラフによる。そのｃｏｎｔｒｉｂｕｔｉｎｇプログラムがまだ評価中である結果の要素にアクセスを試みる表現は、プログラムが完了するまでブロックする。　フューチャーは、結果としての並行アルゴリズムを実施するのに非常に適したオペレーションの良い例である。ＭｕｌｔｉＬｉｓｐあるいはＭｕｌ－Ｔの表現によって生成されたオブジェクト（フューチャーＥ）は、計算Ｅのためのスレッドを生成する。そしてリターンされたオブジェクトはフューチャーとして知られている。結果としてｖを生じてＥが終了したとき、フューチャーが確定したと言う。フューチャーにタッチする表現は、Ｅがまだ計算されている場合にはブロックし、他方、フューチャーが確立した場合にはｖを与える。

【０２０５】図１１に示す素朴なソーティング・プログラムでは、フューチャーの各例は新しいスレッドの生成を伴う。この振舞は望ましいものではない。それは、プロセス・ツリーのレベルｉで計算を行うフューチャーはレベルｉ＋１などにおいてその子に対して明らかなデータ依存性を有しているといった理由による。このプログラムにおいてデータ依存性があった場合、プロセッサおよび記憶装置の利用度が低下する結果となる。これは、生成されたライトウエイトのプロセスの多くが、まだ未評価のフューチャーのものとして他の値をリクエストするときブロックする必要があるか、または、例えば、小さい素数を計算するプロセスの場合、それらを生成するために必要なコストに比べ、少量の計算を行うためである。

【０２０６】スレッドの動的な状態は大きいオブジェクト（例えば、スタックおよびヒープ）から成るので、プロセスのブロッキングが頻繁に生じる場合、あるいはプロセスにグラニュラリティが小さすぎる場合、キャッシュおよびページのローカリティについては妥協する。

【０２０７】タッチおよびフューチャーの意味規則は、他のフューチャーＧにタッチするフューチャーＦは、Ｇがまだ確定していない場合、Ｇでブロックしなければならないということを命令する。T_f および T_e をそれぞれＦおよびＧのスレッド表現とする。Ｇでのタッチ・オペレーションのランタイム・ダイナミックスは、ＴＢが（ａ）遅延またはスケジュールされたとき、（ｂ）評価しているとき、（ｃ）または確定したときのいずれかの場合、T_e に対するアクセスを伴う場合がある。最後のケースでは、これらのスレッド間で同期化は不要である。ケース（ｂ）の場合、T_f は T_e が完了するまでブロックする必要がある。ケース（ａ）の場合、スティングでは重要な最適化を行う。これについては以下に説明する。

【０２０８】ＴＦは、ＴＧ内に閉じ込められた閉包（Ｅと呼ぶ）を、コンテクスト・スイッチをブロックし、強制するより、むしろそれ自身のスタックヒープとを用いて評価することができる。実際、スティングでは、Ｅを通常のプロシージャとして扱い、Ｇのタッチを単純なプロシージャ・コールとして扱う。この場合、T_f が T_e を吸収すると言う。T_f は、その他の場合には必然的にブロックするという点でこの最適化は正しい。T_f の動的なコンテクストを用いてＥを適用することによって、T_f が動作するＶＰは、コンテクスト・スイッチを実行するというオーバーヘッドを負わない。また、T_f のＴＣＢが代りに用いられるので、T_e に対してＴＣＢを割り当てる必要がない。

【０２０９】この最適化は、コールしているスレッドが

45

必ずしもブロックする必要がない場合に用いられたとき、目立って異なった結果を導くのみとなる場合がある。例えば、T_c が T_r によるスペキュラティブ・コールの要素であったとする。さらに、T_c は分岐するが、他のスペキュラティブ・スレッド（T_s と呼ぶ）は分岐しないとする。吸収が無い場合には、T_c および T_s は共に別々のスレッド・コンテクストを生む。しかし、吸収がある場合には、T_r は T_c を吸収することができ、そして、T_c がループするので T_r もループしよう。スレッドが吸収できるか、またはできない場合、ユーザはスレッドの状態をパラメータ化して、ＴＣに通知することができる。スティングはこのためのインターフェース・プロシージャを提供する。

【０２１０】吸収のため、スティングはコンテクスト・スイッチィングのオーバーヘッドを低減させ、そしてプログラムにおいてプロセスが互いに強いデータ依存性を示すとき、そのプログラムに対するプロセスのグラニュラリティを増大させる。もちろん、オペレーションを最も効果的なものにするため、スケジュールされたスレッドが吸収された状態になり得るよう、スレッドのグラニュラリティは十分に大きいものでなければならない。プロセスのグラニュラリティが小さすぎる場合には、吸収しているスレッドがそれらの値を要求できる前に、プロセッサは吸収され得る可能性のあるスレッドの評価を開始しよう。

【０２１１】負荷にもとづくインライニングおよびレイジーなタスク生成は、他の並列Ｌｉｓｐシステムに応用された２つの他の同種の最適化である。負荷にもとづくインライニングでは、現在のシステムの負荷がある特定のスレッシュホールドを越えた場合、スレッドはインライン（すなわち、吸収）される。この最適化では、プログラマの介入は不要であるだけでなく、ある種の条件のもとでは、本来終了するはずのプログラムがデッドロックあるいは長時間の停止状態になる場合がある。これはインライニングの決定が撤回できないからである。従ってこの最適化では、タスクが、そのデータ依存性のためにある順序で評価される必要があるとき、それとは異なる特定の評価の順序をタスクに課す。スレッドの吸収は、吸収されない場合にはスレッドがブロックするときのみ、そしてデータの依存性が保証されているときのみ生じるので、この問題の影響を受けない。

【０２１２】レイジーなタスクの生成は、負荷にもとづくインライニングに係わる多くの問題を解決する。レイジーなタスクの生成では、常にすべてのスレッドの評価がインラインされるが、しかしプロセッサがアイドル状態となったとき、このインライニング・オペレーションを撤回可能とする。スレッドは実際に必要とされない限り決して生成されない。この設計ではプログラマの介入を必要とせず、本来デッドロックしないプログラムのデッドロックを招かず、そして、実際に発生されるタスク

46

の数が低減される。

【０２１３】スレッドの吸収はレイジーなタスクと主に２つの点で異なっている。（１）スレッドの吸収は、アプリケーションによって決まるスケジューリング・プロトコルが存在しても働く。レイジーなタスク生成はグローバルＬＩＦＯスケジュールと、インラインされたスレッドを保持するための単一の待ち行列の存在とを仮定する。（２）レイジーなタスク生成は、１つのプロセッサに対して１つのグローバル・ヒープを用いる。レイジーなタスクの生成では、タスクがスティールされたとき、スレッド吸収の場合よりローカリティは低下する。第２に、レイジーなタスク生成の場合のガーベッジ・コレクションでは、システム内のすべてのスレッドを停止させる必要がある（コレクタそれ自身が並列であっても）。スレッドの吸収の場合にはこの制約はない。

【０２１４】他の例はマスタ・スレーブのパラダイムであり、これは並列プログラムを構成するためのポピュラーな技術である。この技術では、発生されたプロセスのコレクションは先験的に行われる。マスタ・プロセスはいくつかのワーカー・プロセスを発生し、それらの結果を結合する。プロセスの通信は典型的には共有並行データ構造あるいは共有並行変数を通じて行われる。マスタ・スレーブ・プログラムがしばしば、ストック・マルチプロセッサ・プラットフォーム上の結果の並列プログラムより効率的である。それは、ワーカーが、それらの結果を発行する場合を除いて、ほとんど互いに通信する必要がないからである。そしてプロセスのグラニュラリティを調整でき、より高い性能が得られる。

【０２１５】スキームにおけるファーストクラスのタプル空間を最適化して実現するためにスティングを用いた。タプル空間は、同期化コンテント・アクセサブル・メモリの抽象体として機能するオブジェクトである。タプル空間は、マスタ／スレーブにもとづく多数のアルゴリズムを具体化するための自然の選択である。

【０２１６】タプルはオブジェクトであり、タプル・オペレーションはバインディング表現であって、ステートメントではないので、ファーストクラスの指示可能なタプル空間の存在により、モジュール性および表現性がさらに向上する。望ましい実施例では、タプル空間は、同期化したベクトル、待ち行列、ストリーム、セット、共有変数、信号、あるいはバッグとして特殊化できる。タプル空間上で許可されたオペレーションは、それらの表示において不変である。さらに、アプリケーションは必要ならタプル空間の間の継承階級を指定できる。

【０２１７】プロセスは新しいタプルをタプル空間に読み込んだり、除去したり、預けることができる。読み込みオペレーションあるいは除去オペレーションにおけるタプル・引数は“テンプレート”と呼ばれ、“？”を前に付けた変数を含むことができる。このような変数は“フォーマル”と呼ばれ、マッチ・オペレーションの結

47

果としてバインディング値を獲得する。これらのフォーマルによって獲得されたバインデイング値は、下位の表現の評価において用いられる。従って、次のように書くことができる。

【０２１８】（ｇｅｔ　ＴＳ［?ｘ］
（ｐｕｔ　ＴＳ［（+ｘ１］））
これによって１つのタプルがＴＳから除去され、１だけインクリメントされ、そしてＴＳに再び預けられる。

【０２１９】この実施例ではまた、スレッド吸収も利用して、タプル空間上で同期するグラニュラリティの細かい並列プログラムの構築を可能とする。スレッドはタプル内で真正な要素として用いられる。次の表現を実行するプロセスＰを考える。

【０２２０】（ｒｄ　ＴＳ［ｘ１　ｘ２］Ｅ）
ここで、ｘ１とｘ２は非フォーマルである。さらに、ＴＳ内のタプルがオペレーション（ｓｐａｕｎ　ＴＳ［E_1　E_2］）の結果として預けられているとする。このオペレーションはE_1とE_2を計算する２つのスレッド（TE_1およびTE_2と呼ぶ）をスケジュールする。TE_1とTE_2が共に完了すると、結果としてのタプルは２つの確定したスレッドを含んでいる。マッチング・プロシージャは、タプル内でスレッドに遭遇したとき、ｔｈｒｅａｄ－ｖａｌｕｅを適用する。このオペレーションはそのスレッドの値を回収する。

【０２２１】しかし、Ｐが実行されるときTE_1がまだスケジュールされている場合には、Ｐはそれを自由に吸収でき、その結果がｘ１に一致するときは確定する。一致するものが存在しない場合には、Ｐは、スケジュールされた状態にあるかもしれないTE_2を残して、他のタプルのサーチへと進む。その後、他のプロセスがこの同じタプルを調べることは可能であり、正当な理由があるならTE_2を吸収する。同様に、TE_1の結果がｘ１と一致するなら、Ｐは次にTE_2を自由に吸収できる。TE_1またはTE_2のいずれかがすでに評価を行っている場合には、Ｐは、１つ（または両方）のスレッドでブロックするか、またはＴＳ内で、他に一致する可能性のあるタプルを調べるかを選択する。タプル空間の意味規則は、この点でこの実施例に対して制約を課さない。

【０２２２】スティングの、ファーストクラスのスレッドとスレッド吸収との組み合せは、共有データ構造を用いて、疑似要求によって駆動されるグラニュラリティの細かい（結果）並列プログラムを書くことを可能とする。この意味で、スレッド・システムは、構造にもとづく同期化（例えば、タプル空間）とデータフロー・スタイルの同期化（例えば、フューチャー／タッチ）との間の意味のある区別の最小化を試みる。

【０２２３】スペキュラティブ並列は重要なプログラミング技術であるが、それを実現した際に生じるランタイムのオーバーヘッドのために、しばしば効果的に用いることができない。スペキュラティブ・プログラミング・

48

モデルをサポートするシステムに最も頻繁に係わる２つの特徴は、他のものより一層有望なタスクを奨励する能力と、不要な計算を中止および再利用（そして、恐らく取消し）する手段を有することである。

【０２２４】スティングは次のことによって、プログラマがスペキュラティブ・アプリケーションを書くことを可能とする。

【０２２５】１．ユーザがスレッドの優先順位を明示的にプログラムすることを可能とする。

【０２２６】２．他のスレッドが完了したとき、あるスレッドがウエイトできるようにする。

【０２２７】３．スレッドが他のスレッドを終了させることを可能とする。

【０２２８】優先順位をプログラムできるので、有望なタスクはそうでないものより先に実行することができる。タスクの組の中で最初に終了するタスクαは、その終了の際、ブロックされているスレッドをどれでも目覚めさせることができる。この機能によって、スティングはＯＲ並列の有用な形態をサポートできる。タスクαは、そのタスクの組の中の他のタスクはすべて、それらの結果が不要であると確定されたなら、終了させることができる。しかし、スティングを用いた理論的計算は、不要なタスクによってもたらされたノンローカルな副作用を取消すことはできないであろう。このシステムは基本的な逆戻りのメカニズムは提供しない。

【０２２９】ｗａｌｔ－ｆｏｒ－ｏｎｅコンストラクトを実現することを考える。このオペレータは、並行してこの引数のリストを評価し、その最初の引数によって生成された値を復帰させ、終了する。従って、表現（ｗａｌｔ－ｆｏｒ－ｏｎｅ　$a_1 a_2 \ldots a_i \ldots a_n$）において$a_i$からｖが生じた場合、この表現はｖを復帰させ、そして、プログラマが必要とするなら、残っているすべてのa_j，$j \neq 1$の評価を終了する。

【０２３０】ＡＮＤ並列を実現したｗａｌｔ－ｆｏｒ－ａｌｌコンストラクトの仕様の同様である。これも並行してその引数を評価する。ただしすべての引数を終ったときのみ真を復帰させる。従って表現（ｗａｌｔ－ｆｏｒ－ａｌｌ　$a_1 a_2 \ldots a_i \ldots a_n$）は、この表現を実行するスレッドはすべての$a_i$が終るまでブロックされているので、障壁同期化ポイントとして機能する。このオペレーションの実現は、スペキュラティブｗａｌｔ－ｆｏｒ－ｏｎｅオペレーションの実現と非常に似ている。

【０２３１】ＴＣはこれらのオペレーションを、共通プロシージャであるｂｌｏｃｋ－ｏｎｓｅｔを用いて実現する。スレッドおよびＴＣＢは、この機能をサポートするように定義されている。例えば、ＴＣＢ構造体に関連しているのは、ＴＣＢの関連するスレッドが再開できる前に終了しなければならない、グループ内のスレッドの数に関する情報である。

【０２３２】ｂｌｏｃｋ－ｏｎ－ｓｅｔは、スレッドのリストとカウントを取る。これらのスレッドは、上述したｗａｉｔ－ｆｏｒ－ｏｎｅオペレーションおよびｗａｉｔ－ｆｏｒ－ａｌｌオペレーションの引数に対応している。カウントの引数は、現在のスレッド（すなわち、ｂｌｏｃｋ－ｏｎ－ｓｅｔを実行しているスレッド）が再開を認められる前に終了しなければならないスレッドの数を表している。この数が１の場合、結果はｗａｉｔ－ｆｏｒ－ｏｎｅを実現したものであり、上記数がｎの場合、結果はｗａｉｔ－ｆｏｒ－ａｌｌの実現である。

【０２３３】組の中のスレッドT_aと、Ｔを待つべき現在のスレッド（T_a）との関係は、下記のものに対する参照を含むデータ構造（スレッド・バリア（ＴＢ）と呼ばれる）内で維持される。

【０２３４】１．T_aのＴＣＢ

２．T_a上でブロックされている他のウエイターのＴＢ

（存在する場合）ｂｌｏｃｋ－ｏｎ－ｓｅｔを定義するプログラムを、図１２に示す。

【０２３５】次のコール

（ｂｌｏｃｋ－ｏｎ－ｓｅｔ　ｍ　T_1 T_2 . . . T_a）

＊は現在のスレッド（Ｔと言う）に、ｍ個のT_i（$m \leq n$）が終了したときアンブロックさせる。これらT_iのそれぞれは、それらのウエイターのチェーン内にＴに対する参照を有している。

【０２３６】アプリケーションはｂｋｏｃｋ－ｏｎ－ｓｅｔを、アプリケーションが終了したときa_iによって起動されるプロシージャｗａｋｅｕｐ－ｗａｌｔｅｒｓと共に用いる。ｗａｋｅｕｐ－ｗａｌｔｅｒｓは、そのスレッド引数内のウエイター・スロットから、連鎖状のウエイターのリストを調べる。ウエイト数がゼロになるウエイターは、いずれかのＶＰのレディー待ち行列に挿入される。ＴＣは、スレッドＴが終了したときはいつもｗａｋｅｕｐ－ｗａｌｔｅｒｓを起動する（例えば、Ｔが終了したとき、または異常に存在するときはいつも）。Ｔの終了を待っているスレッドは、すべてこのようにしてリスケジュールされる。

【０２３７】これは２つのプロシージャが与えられると、ｗａｉｔ－ｆｏｒ－ｏｎｃは次のように簡単に定義することができる。

【０２３８】

```
(define (wait-for-one . block-group)
   (block-on-group 1 block-group)
   (map thread-terminate block-group
)
```

Ｔがｗａｉｔ－ｆｏｒ－ｏｎｅを実行する場合、それはｂｌｏｃｋ－ｇｒｏｕｐ引数内のすべてのスレッド上でブロックする。Ｔが再開されるとき、Ｔは、利用できるいずれかの仮想プロセッサのＴＰＭ内のレディー待ち行列に配置される。Ｔの再開のとき実行されるマップ・プロシージャは、そのグループ内のすべてのスレッドを終了させる。

【０２３９】スティングのプロシージャｗａｉｔ－ｆｏｒ－ａｌｌは、このオペレーションを省略できる。それは、そのブロック・グループ内のすべてのスレッドは、このオペレーションを実行するスレッドが再開される前に、終了することが保証されているからである。

【０２４０】スティングは、８プロセッサのＳｉｌｉｃｏｎ Ｇｒａｐｈｉｃｓ Ｐｏｗｅｒ Ｓｅｒｉｅｓ（ＭＩＰＳ Ｒ３０００）と、１６プロセッサのＳｉｌｉｃｏｎ Ｇｒａｐｈｉｃｓ Ｃｈａｌｌｅｎｇｅ（ＭＩＰＳ Ｒ４４００）の両方において実現した。両マシンは、共有（キャッシュ・コヒーレント）マルチプロセッサである。この抽象物理的マシン構成では、物理的プロセッサはライトウエイトのＵｎｉｘスレッドにマッピングされる。マシン内の各プロセッサは、このようなスレッドの１つをランさせる。

【０２４１】以上、コンピュータ・ソフトウエア・アーキテクチャの望ましい実施例について記述し、説明したが、当業者にとって明らかなように、本発明の広範な原理および趣旨から逸脱することなく、種々の変形や変更を加えることは可能である。

【０２４２】

【発明の効果】以上説明したように本発明によれば、高度並列マルチプロセッサ／マルチコンピュータ・システムを制御するための、現代のプログラミング言語に対する非常に効率の良いサブストレートとして役立つコンピュータのオペレーティング・システム・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式が得られる。

【０２４３】更に本発明によれば、カスタマイズ可能な仮想マシンにもとづく非同期の計算のためのソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式が得られる。

【０２４４】また本発明によれば、仮想プロセッサ上でファーストクラスのオブジェクトとしてライトウエイト・スレッドをサポートするソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式が得られる。

【０２４５】更に本発明によれば、カスタマイズ可能なポリシー・マネージャを、特にユーザ・レベルに含むソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式が得られる。

【０２４６】また、本発明によれば、カスタマイズ可能な仮想トポロジーを含むソフトウエア・アーキテクチャ

51

を用いた高度並列コンピュータ・システムの制御方式が得られる。

【０２４７】更に本発明によれば、スレッド吸収、遅延ＴＣＢ割り当て、ならびに記憶装置共有の場所としてのスレッド・グループを含むソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式が得られる。

【０２４８】また本発明によれば、多様な形態のポートを含むソフトウエア・アーキテクチャを用いた高度並列コンピュータ・システムの制御方式が得られる。

【０２４９】更に本発明によれば、上述のようなソフトウエア・アーキテクチャを用いて制御されるコンピュータ・システムが得られる。

【図面の簡単な説明】

【図１】本発明の一実施例によるソフトウエア・アーキテクチャを用いた制御方式を示すブロック図である。

【図２】図１の抽象物理的マシンおよび仮想マシンを示す図である。

【図３】本発明のオペレーティング・システムの抽象アーキテクチャを示す概略ブロック図である。

【図４】本発明で用いるスレッドの状態およびＴＣＢの状態の遷移を示す図である。

【図５】本発明で用いる記憶装置の構成を表す概略図である。

【図６】本発明で用いるスレッドのプログラミングを説明するためのプログラムを示す図である。

【図７】本発明で用いる物理的プロセッサの２Ｄメッシュ上で多重化された仮想プロセッサの３Ｄメッシュを生成するプログラムを説明するための図である。

【図８】本発明で用いるコンテクスト・スイッチを始動するプログラムを示す図である。

【図９】本発明で用いるコンテクスト・スイッチを終了するプログラムを示す図である。

【図１０】本発明で用いる新しいスイッチを開始するプログラムを示す図である。

【図１１】本発明で用いるグラニュラリティの細かい適応並列ソート・アルゴリズムのための最上位のプロシージャのプログラムを示す図である。

【図１２】本発明で用いるｂｌｏｃｋ－ｏｎ－ｓｃｔを定義するプログラムを示す図である。

52

【符号の説明】

１０　抽象物理的マシン
１１　物理的トポロジー
１２　抽象物理的プロセッサ
１３　仮想プロセッサ・コントローラ
１４　仮想マシン
１５　仮想プロセッサ・ポリシー・マネージャ
１６　仮想プロセッサ
１７　スレッド・コントローラ
１８　スレッド
１９　スレッド・ポリシー・マネージャ
２０，２０′　仮想トポロジー
２４　仮想マシン／アドレス空間
２６　グローバル記憶プール
２８　グローバル共有オブジェクト
３０　ルート環境
３１　スタック
３２　ＴＣＢ
３３　ローカル・ヒープ
３５　グローバル・ヒープ
３６　遅延
３８　スケジュール
４０　評価
４２　吸収
４４　確定
４６　初期化
４８　レディー
５０　ラン
５２　ブロック
５４　保留
５６　終了

【図６】

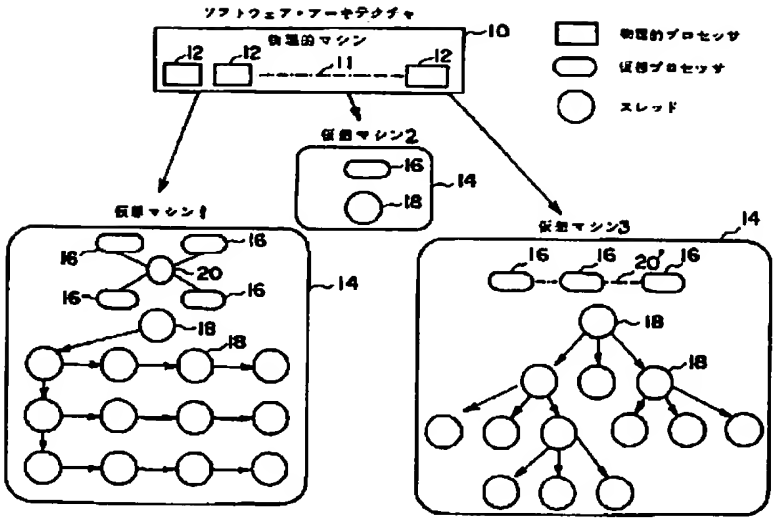
```
(define (filter op n input)
  (let loop ((x (hd input))
             (output (make-stream))
             (last? true))
    (cond ((zero? (mod x n))
           (loop (rest input) output last?))
          (last?
            (op (lambda () (filter op x output)))
            (loop (rest input) (attach x output) false))
          (else (loop (rest input) (attach x output) last?)))))

(define (sieve op n)
  (let ((input (make-integer-stream n)))
    (op (lambda () (filter op 2 input)))))
```

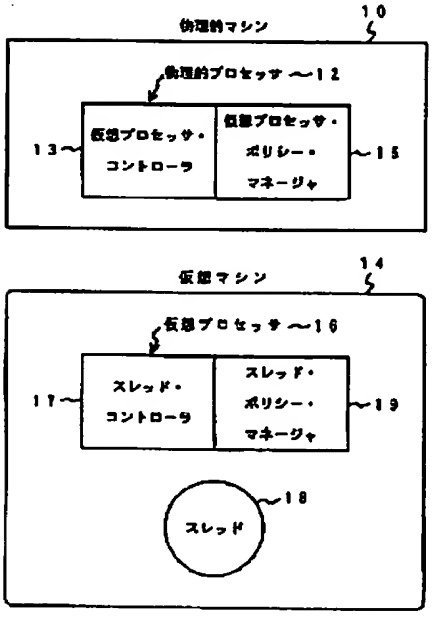
【図７】

```
(define (create-3D-mesh depth)
  (let* ((pe-width (get-pe-width))
         (pe-height (get-pe-height))
         (3D-mesh (make-array w h depth)))
    (let ~i~ ((i 0))
      (if (< i pe-width)
          (let ~ej~ ((j 0))
            (if (< j pe-height)
                (let ((vp (create-vp (get-pe i j))))
                  (set-vp-address vp (vector i j))
                  (let ~ek~ ((k 0))
                    (cond ((< k depth)
                           (set-aref! 3D-mesh vp)
                           (~ek~ (+ k 1)))))
                  (~ej~ (+ j 1)))))))))
```

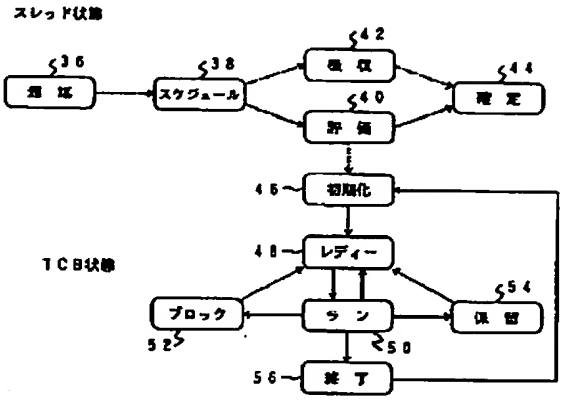
【図1】



【図2】



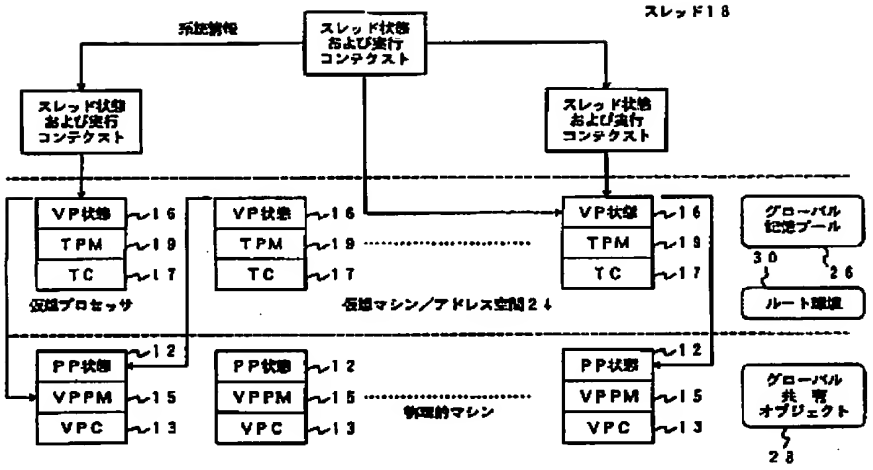
【図4】



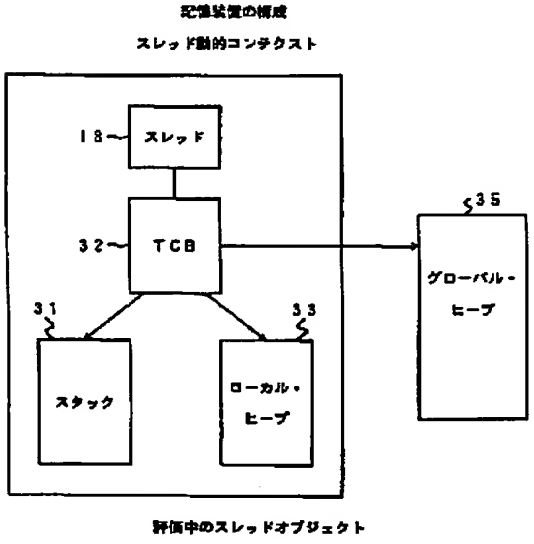
【図9】

```
(define (finish-context-switch)
  (let* ((vp          (current-vp))
         (previous    (vp.current-tcb vp))
         (state       (tcb.state previous)))
    (set-vp.current-tcb vp (current-tcb))
    (cond ((null? previous (current-tcb))
           (cond ((tcb-state-ready? state)
                  (if (null? previous (vp.root-tcb vp))
                      (tpo-enqueue-ready-thread vp previous)))
                 ((tcb-state-blocked? state)
                  (thread-cutex-release (tcb.thread previous)))
                 ((tcb-state-suspended? state)
                  (tpo-enqueue-suspended-thread vp (tcb.thread previous))
                  (thread-cutex-release (tcb.thread previous))))))
    (enable-preemption)))
```

【図３】



【図５】



【図８】

```
(define (start-context-switch next-state)
  (disable-preemption)
  (let* ((vp (current-vp))
         (next (let ((obj (tpm-get-next-thread vp)))
                 (if (false? obj)
                     (if (tcb-state-ready? next-state)
                         (current-tcb)
                         (tpm.vp-idle vp))
                     obj))))
    (cond ((eq? next (current-tcb)))
          ((tcb? next)
           (set-tcb.state next tcb-state/running)
           (set-thread.vp (tcb.thread next) vp)
           (cond ((eq? next-state tcb-state/dead)
                  (return-current-tcb-to-vp-pool vp)
                  (restore-tcb-and-registers next))
                 (else
                  (save-current-tcb-registers)
                  (restore-tcb-and-registers next))))
          ((thread? next)
           (thread-mutex-acquire next)
           (let ((tcb (allocate-tcb next-state vp)))
             (setup-new-thread next vp tcb)
             (if (eq? tcb (current-tcb))
                 (save-current-tcb-registers))
             (thread-mutex-release next)
             (start-new-tcb tcb start-new-thread))))))
```

【図１１】

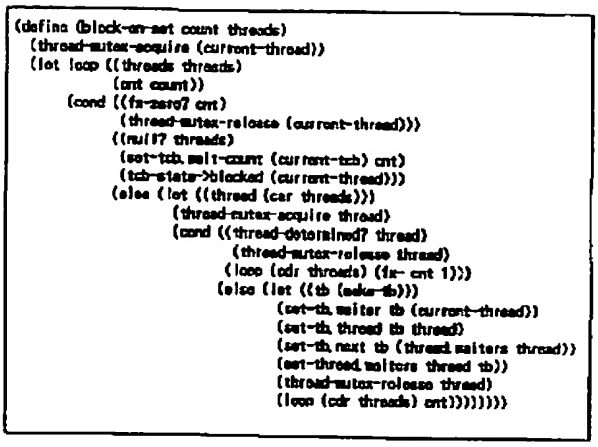
```
(define (pbisort root spare up?)
  (cond ((node-left root)
         (let ((left-half (future (pbisort (node-left root) root up?))))
           (pbisort (node-right root) spare (not up?))
           (touch left-half)
           (pbimerge root spare up?)))
        (else (compare-and-swap root spare up?))))

(define (pbimerge root spare up?)
  (let loop ((root root) (spare spare))
    (if (compare-and-swap root spare up?)
        (flip-tree-1 root up?)
        (flip-tree-2 root up?)
    (cond ((node-left root)
           (let ((left-half (future (loop (node-left root) root))))
             (loop (node-right root) spare)
             (touch left-half))))))
```

【図１０】

```
(define (start-new-thread)
  (let ((z (error-value "Thread has no value")))
    (unwind-protect (set z (catch exit
                             (set-tcb.exit-handler (current-tcb) exit)
                             ((thread.thunk (current-thread)))))
      (let ((thread (current-thread)))
        (thread-go thread)
        (set-thread.value thread z)
        (make-up-waiters thread)
        (set-tcb.state (current-tcb) tcb-state/dead)
        (start-context-switch tcb-state/dead)))))
```

【図１２】



```
(define (block-on-set count threads)
  (thread-mutex-acquire (current-thread))
  (let loop ((threads threads)
             (cnt count))
       (cond ((fx=zero? cnt)
              (thread-mutex-release (current-thread)))
             ((null? threads)
              (set-tcb.wait-count (current-tcb) cnt)
              (tcb-state->blocked (current-thread)))
             (else (let ((thread (car threads)))
                     (thread-mutex-acquire thread)
                     (cond ((thread-determined? thread)
                            (thread-mutex-release thread)
                            (loop (cdr threads) (fx- cnt 1)))
                           (else (let ((tb (make-tb)))
                                   (set-tb.waiter tb (current-thread))
                                   (set-tb.thread tb thread)
                                   (set-tb.next tb (thread.waiters thread))
                                   (set-thread.waiters thread tb))
                                   (thread-mutex-release thread)
                                   (loop (cdr threads) cnt))))))))
```